THE PROVISION OF RELOCATION TRANSPARENCY THROUGH A FORMALISED NAMING SYSTEM IN A DISTRIBUTED MOBILE OBJECT SYSTEM

By

Katrina Elizabeth Falkner, B.Sc.(Ma. & Comp. Sc.)(Hons)

September 22, 2000

A THESIS SUBMITTED FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

IN THE DEPARTMENT OF COMPUTER SCIENCE

UNIVERSITY OF ADELAIDE

Contents

List of Tables

List of Figures

Abstract

Mobility in distributed object systems is useful as it can provide such properties as load balancing, code to data movement, fault tolerance, migration to stable storage, and autonomous semantics. In a widely distributed system, these properties are important as they can help alleviate latency issues and increase performance within the system. Additionally, they provide more flexibility in the programming of distributed systems by relaxing static location restrictions. Location transparency removes the need for client objects to explicitly know or define the location of a server object when communicating. If a server object is capable of migration, relocation transparency maintains reference validity throughout the migration.

Several models for providing relocation transparency exist, including the home location, forwarding location, and broadcast models. This thesis proposes a model that uses a distributed registry system and dynamic reference updating to provide location and relocation transparency. A registry system is used to provide location independence by resolving a location independent *name* to a reference that can be used by a client. A naming system is used to provide correct binding and production of names within the naming restrictions of the system.

The thesis proposes that the choice of naming system within a distributed or mobile object system has a large effect on the system's ability to support efficient transparent object relocation. This thesis proposes that a formal analysis of naming systems enables the selection of an appropriate naming system for a distributed or mobile object system given the object system's naming, distribution and transparency requirements. This thesis presents a new classification scheme for naming systems, based on analysis of a broad spectrum of naming systems.

A classification of existing mobile and distributed object systems with respect to existing naming models is provided. It is shown that the current models need to be refined and extended to completely and correctly classify the example systems. This thesis proposes extensions and refinements that enable correct and complete classification of mobile and distributed object systems with a need for transparency. The extended naming model is then used to describe a naming system that is capable of implementing any naming system classifiable by the extended model. A classification of a naming system to support the proposed model of location and relocation transparency is presented.

A distributed ORB system is designed and implemented to support the distributed namespace and generic naming system implementation. The distributed ORB system is hierarchically structured and is capable of adapting in response to node failure. This ORB system is used to support client and server object integration in the DISCWorld metacomputing environment. The ORB system is used to provide migration, replication and cloning services to the DISCWorld metacomputing environment.

A qualitative analysis of the generic naming system and the DISCWorld ORB system is performed. A comparison between the proposed model for location and relocation transparency and existing models is also presented. This comparison shows that the proposed model exhibits better location and relocation performance within the DISCWorld environment. The distributed nature of the ORB system and namespace provides a scalable nature in terms of namespace size, the number of objects within the system, and the frequency of location and relocation requests.

Declaration

This work contains no material which has been accepted for the award of any other degree or diploma in any university or other tertiary institution and, to the best of my knowledge and belief, contains no material previously published or written by another person, except where due reference has been made in the text.

I give consent to this copy of my thesis, when deposited in the University of Adelaide Library, being available for loan and photocopying.

Although DISCWorld is a joint development of the Distributed & High Performance Computing Project group and other students, the work reported here on transparent mobility support and naming systems within DISCWorld is my own work.

Katrina Elizabeth Falkner, B.Sc.(Ma. & Comp. Sc.)(Hons)
September 22, 2000

Acknowledgments

This work was carried out under the Distributed High Performance Computing Infrastructure Project (DHPC-I) of the On-Line Data Archives Program (OLDA) of the Advanced Computational Systems (ACSys) Cooperative Research Centre (CRC) and funded by the Research Data Networks (RDN) CRC. ACSys and RDN are funded by the Australian Commonwealth Government CRC Program.

I would like to thank my supervisors Michael Oudshoorn and Ken Hawick for their comments on this work and their advice throughout my time as a PhD student. Michael Oudshoorn deserves special thanks for all of his help, especially with the preparation of this thesis, and his comments and interest in my work from when I was an Honours student until now.

Many thanks to Chris Barter, Professor at Adelaide University, for his support throughout my time at Adelaide University. Thanks also to Paul Coddington, Kevin Maciunas, Dave Munro, Cheryl Pope, Francis Vaughan and Tracey Young.

Thanks also to past and present members of the DHPC group, who have made my time as a PhD student an enjoyable experience: Duncan Grove, James Hercus, Heath James, Kim Mason, Jesudas Mathew, Craig Patten, Andrew Silis, Darren Webb and Andrew Wendelborn.

I would like to thank my family for their constant support throughout my studies. My parents believe that, with education, a person is able to do anything they wish. With their help and support, and the guidance of my sister Julie, this has been possible for me.

Most of all, I would like to thank my husband, Nick Falkner, whose support and friendship has been invaluable. Providing me with a fantastic and supportive environment, reading my thesis drafts, and keeping me motivated were just part of the daily routine.

Katrina Falkner.
September 22, 2000.
Adelaide, Australia.

Chapter 1

Introduction

The object model is one of the most powerful and commonly used programming paradigms. Distributed object systems require objects to remotely communicate, performing coordinated computation. Migration of objects is an important aspect of distributed object computing as it can help alleviate the effect of object or node failures, and communications latency.

A *mobile object* is an object that is capable of moving, or being moved, readily from one place to another. Mobility within a distributed object system allows mobility-enabled objects to move freely between processing nodes within the distributed system. Mobility within a distributed object system supports:

- load balancing (by moving objects from heavily loaded nodes to lightly loaded nodes),
- fault tolerance (by moving objects to new nodes in response to node partial failure),
- locality of data access (by moving objects to the vicinity of the data they require) and
- scheduling and monitoring of the distributed system (by allowing monitoring objects to move about the distributed system assessing load and scheduling requirements).

Mobility can be used to hide latency within a widely distributed system by providing specialised communications styles [45], such as moving an object to a remote site to perform computation and then recalling the object to extract result values in place of a traditional remote invocation in client/server systems. Additionally, mobility in a distributed object system allows flexibility in the programming of distribution by removing static location definition restrictions.

Mobility has been examined at many different levels: from thread or task-level mobility within multiprocessor systems [102], and process or object mobility within a distributed system [128, 130] to host mobility within a disconnected network [6, 93, 146]. One of the most important issues when dealing with mobility within an object system is that of locality and reference management [129,130,158]. Objects within a mobile object system may have references to other objects within the system. When these objects move, it is important that any references to them remain valid and that any referencing object always be able to contact the mobile object regardless of the mobile object's location.

An object reference that does not expose the location of the referenced object is said to be *location transparent*; a location transparent reference can access a remote object in the same manner regardless of the remote object's current location or state. Location transparent references have been implemented in several distributed object systems [83, 86, 109] and allow distributed communication to be programmed without exposing or requiring location specification. Transparency decreases complexity in distributed programming by taking away the need to define object locations, but incurs additional communications overhead as the location transparent reference must be resolved or evaluated to a direct reference to the remote object [47, 113, 162].

As stated by Shapiro *et al* [164], *"it is essential for performance that a stub contain the actual address of its target"*, where a stub is a type of reference. However, this requirement leads to difficulties when the embedded location becomes invalid through object migration. An object reference that does not require knowledge of the mobile object's location is said to be *location independent*.

An object that can change location without affecting other objects is said to be *relocation transparent*. In a relocation transparent system, a client does not have to explicitly update its references in any way as all relocation is performed transparently. An object that can additionally change location without performing an explicit migratory action is termed *migration transparent*.

An additional problem in a distributed object system is that of initially obtaining the location of, or a reference to, an object. This task, and that of maintaining a reference, is often performed by *naming* an object in some way and making that name known to potential referencing objects. Saltzer states that *"Names for objects are required so that programs can refer to the objects, so that objects can be shared, and so that objects can be located at some future time."* [158]. A name can take the place of, or be a part of, an object reference. This causes the location transparency and location dependence of the reference to be dependent on the name's location transparency and location dependence. For example, suppose an object reference is a combination of its object or procedure name, its host and optionally some unique identifier for that host. This produces a globally unique name, however it is neither location transparent or location independent.

The mechanism of matching a name to an object is termed *name binding*; the mechanism of obtaining an object or object reference from a name is termed *name resolution*. A *naming context* is the domain in which the names are understood. A name is always resolved with respect to some naming context. A naming context may be a local context such as a single node or a cluster, or a global context may be defined. A *naming model* defines how binding and resolution is to be performed and how strong or weak restrictions on these actions will be. A naming model is implemented by a *naming system*.

Many distributed object systems utilise a global or local naming service of some form to identify services registered within the system. This naming service provides the ability to locate a service from a name, to link or bind a service to a name and potentially to track a service once linked to a name. Distributed object systems do not have an inherent need for transparent names, as once a name has been resolved or a reference has been established, the reference is indefinitely correct unless server failure or shutdown occurs. In this form of system it is difficult to support mobility as references are neither location or relocation transparent.

Many mobile object systems do not support a naming service or relocation transparency and, as a result, provide no way of registering or tracking the identity of a mobile object. If it is to maintain consistency, a mobile object system that allows external references to its objects has a requirement for a globally unique name that is immutable for the lifetime of the object. Once an object has been created and other objects have references to it the name of the object must be globally unique to permit tracking throughout the object system. A mobile object system that wishes to provide this, and similar, forms of consistency and also provide location and relocation transparency must provide a naming system of some kind to manage its namespace.

This thesis proposes a model of location and relocation transparency that uses a distributed naming system. An expressive naming model is used to resolve names to independent, location transparent references. The naming system is also used as a framework that provides mobility in a location and migration transparent manner. This framework can be extended to provide other services such as transparent replication and service cloning. Names in this model act as object references and have variable, hierarchical contexts, leading to some names being part of a global context and others part of a local context.

Binding of names to objects was initially studied in depth by Saltzer [158, 159] with respect to naming objects within operating systems (at the level of addressing schemes to support dynamic binding and file systems)[1]; and has been examined in the implementation of many mobile object systems [41, 62, 102, 162]. Bayerdorffer [15, 16] defines a model of name binding that is applicable to concurrent object systems. This model defines several properties that can, additionally, be used to classify distributed and mobile object systems. Bowman *et al* [26] present a formal model of name resolution. These models are defined in more detail in Chapter 3, and are used to classify existing distributed and mobile object systems according to their name binding and resolution schemes in Chapter 4.

[1]A detailed study of naming and binding relative to a distributed directory structures with focus on client caching is given by Terry in [174].

The classifications presented in Chapter 4 are used to highlight deficiencies in the formal models and outline additional elements or changes to the models that increase their suitability for classifying the example systems. Chapter 5 presents the defined extensions to the models and uses these extensions to classify pertinent systems and to define the extended naming model used within this thesis.

Distributed name services have been studied in the context of distributed file systems [35, 158] and global distribution systems [112]. The need for scalability in the name service requires the name service to be distributed over the processing nodes within the system rather than be centralised [174], however this distributed nature introduces coherency and update problems [35, 112]. These issues have been discussed by Lampson in great detail [112]; additionally by Cheriton and Mann [35] and Terry [174]. A centralised name service for a widely distributed object system introduces a central point of failure and is also a bottleneck for communications; a distributed name service alleviates these problems but forces the system to either pass around large amounts of data (to keep replicas of a logically centralized name service) or to pass requests through some hierarchy. The design issues involved in designing a scalable and coherent distributed name service form part of this thesis.

This thesis explores support for object mobility within distributed object systems, specifically that found in systems based on an Object Request Broker (ORB) model. An ORB system is a form of distributed object system that utilises an intermediate well-known broker object to provide and manage references on a small scale. An ORB does this by providing mappings between names and objects, supporting facilities for name binding and name resolution. ORBs are an interesting platform as the model is easily extensible and provide mechanisms for integrating additional support services into a distributed or mobile object system, such as security or trading services [136].

This thesis extends the naming models provided by Bayerdorffer [15, 16] and Bowman *et al* [26] to be more suitable for classifying distributed object systems with mobility. New characteristics applicable to the classification of distributed and mobile object systems are defined and existing systems are classified according to this extended model. An expressive naming model based on these extended characteristics is outlined and forms the basis for an implementation of location and relocation transparency within a distributed ORB system. An implementation of a distributed ORB using this expressive naming model to support object mobility and location and relocation transparent references is described. Analysis of this implementation shows that the additional overhead in maintaining transparency is outweighed by the benefit of robust reference management and the scalability of the extended naming model and the distributed ORB system.

1.1 Models for Relocation Transparency

Several solutions have been suggested to maintain the validity of existing references within a distributed object system. Mechanisms have been proposed based on the concept of a *home* location; a home location is updated whenever an object moves and serves as the client contact point for referencing. A home location can also be responsible for a form of "descriptive" naming in that it keeps track of any additional information or status, such as whether the object is idle, busy or blocked. The home location model has been used to support location transparency in [12, 34, 49].

One common extension to the home location idea to to use a mobile home location [61, 116]. This results in a chain of homes or *forwarding* locations leading from the original home location to the current location of the mobile object. A client reference may point to any link in the chain. The forwarding location model has been used in [4, 6, 84].

A fragmented object [46, 122, 123] model is often chosen as the distribution model in a distributed object system. A fragmented object is one with several components that may be distributed over nodes within a distributed system, while appearing to be a single entity to external clients. A fragmented object consists of several types of component:

- a set of fundamental components that make up the application,
- client interface components, existing on the client's system, that support communication within the fragment,
- an interface between fragments called the group interface, and
- low-level shared fragments, known as connective objects, that enable communication between other fragments.

By having the server object distributed over several nodes, including client nodes, the client can access the server abstractly regardless of the location of the fundamental components. Multiple client interface fragments are distributed to the sites of clients to allow this form of fragmented access. A client interface fragment can then be used as a named reference and is responsible for maintaining reference validity and for any transparent or independent locality that the model may support.

Fragmented objects are similar to concepts used in RPC [20, 22], CORBA [138], DCOM [27] and Java RMI [167]. The fragmented object model differs from these in that it provides support for connective objects and group interfaces that enables multiple levels of fragmentation, while the other systems provide only one level of fragmentation: separating client interface fragments and the remaining service fragments. Fragmented objects also differ in that they can provide multiple client interface fragments dependent on the client.

A client interface fragment is often termed a *stub* [22, 138, 167] and can be viewed as a simple proxy of the server object within a client/server communication. The stub has

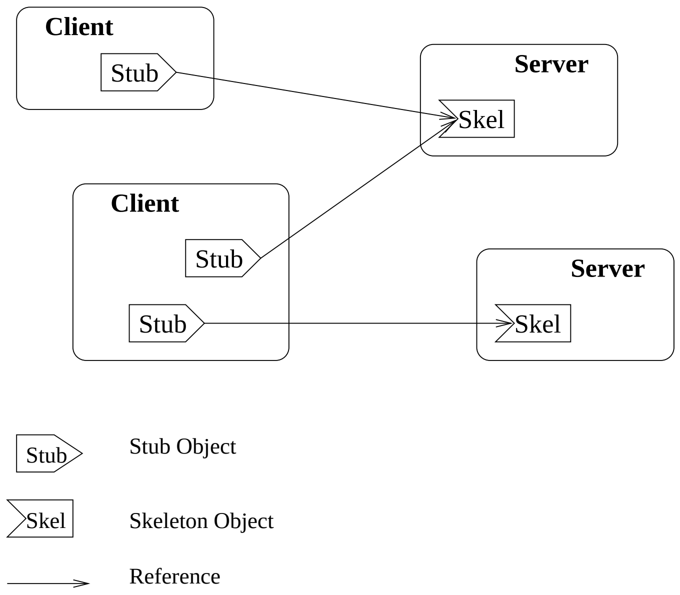


Figure 1: A stub-skeleton style communication.

identical method signatures to the server and, hence, provides an abstract local interface for all server methods available to the client. The stub is responsible for performing the actual remote invocation and also may be responsible for maintaining reference transparency and/or reference independence. On the server side additional code (often termed a skeleton [22, 70, 83]) is used to translate the remote invocation from the stub to a local invocation on the server object. Figure 1 shows a typical client/server system that uses stubs and skeletons to perform remote invocation.

A further alternative used to manage location referencing is that of stub-scion pairs [164] (SSPs). A stub is maintained as the object reference on the client side and a scion is maintained for each stub on the server side; a stub-scion pair is produced for each object reference created. When an object reference is moved, a new stub-scion pair is created to produce a chain of pointers through which the correct reference can be obtained.

The way in which the stub-scion technique differs from that of forwarding locations or a home location is that a stub within a stub-scion system may have multiple references to the referenced object. A stub will contain a *strong locator* reference which will always lead to the destination object and potentially multiple *weak locator* references that are not guaranteed to complete but may contain a shorter path. A strong locator references through the chain of SSPs which will eventually resolve to the server object. A weak reference will directly reference the last known location of the server, encouraging efficient communication. Once the server has relocated, the weak reference become invalid. A weak locator may be returned as part of an invocation through a strong locator reference, and can be used as a direct path for future invocations.

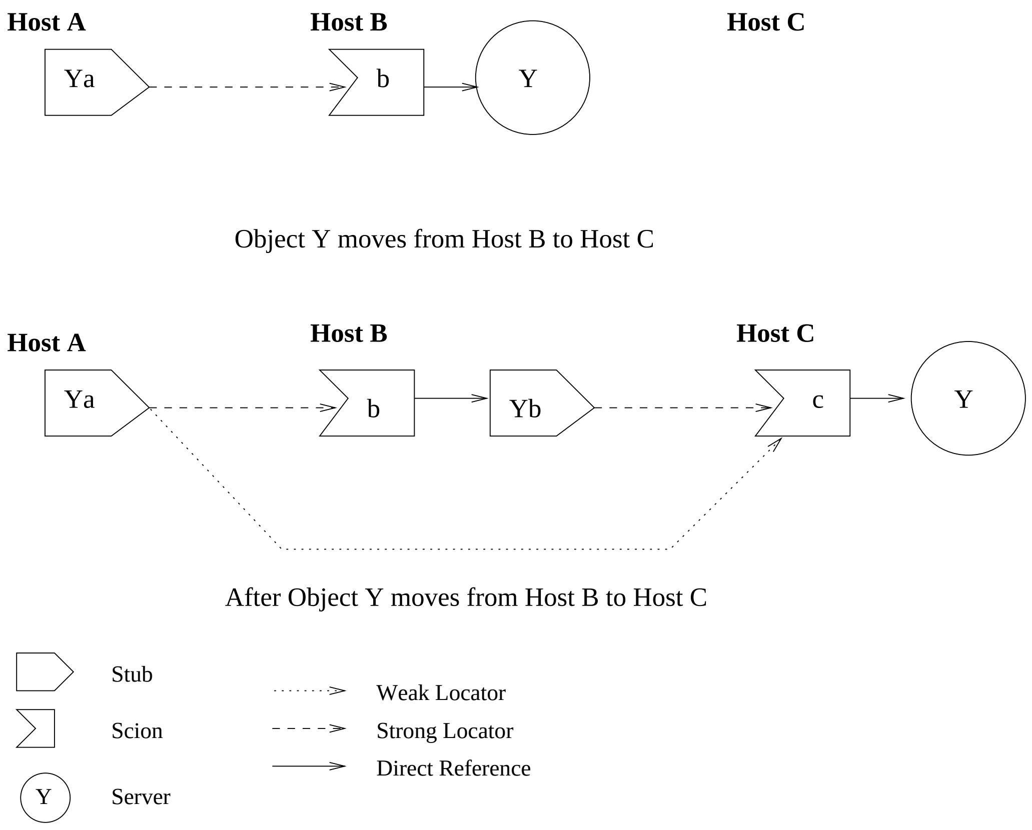


Figure 2: Reference management using stub-scion chains.

Figure 2 shows an example of the stub-scion mechanism where Object Y relocates from Host B to Host C, leaving updates of its location in the form of a SSP chain. Instead of a single forwarding object or home object, scion objects are maintained for each existing reference enabling a form of reference counting. Shapiro *et al* [164] use the stub-scion mechanism to support a distributed acyclic garbage collection system where the invocation protocol is used to provide suggestions of weak locators. Stub-scion chains have been used to support reference management in Hobbes [121] and work by Baggio [7].

A central registry has also been proposed by systems such as Gardens [154], which treats a distributed system as a closely linked parallel system. Each reference is treated as if it were part of a shared memory system where offsets are managed as part of reference access; this results in additional overhead but removes any required reference updating and also any naming requirement. However, this type of system is only suitable for a restricted set of distributed object systems where the structure of the system is known. V-System [124] introduces a search-based mechanism where references are constructed of (*logical host id*, *local index*) pairs. When an object moves, the logical host is duplicated and then set to the new physical host address. As existing references become invalid, their entry in the mapping cache becomes invalid and a request is broadcast to the network for the new logical host id.

1.2 A Proposed Model for Relocation Transparency

The model for location and relocation transparency proposed in this thesis relies on a distributed and scalable registry system which acts as both a mobility service and a naming service.

Location transparency is provided through the use of a distributed naming service. A fragmented object model is used, with some location dependent information kept within the client interface fragment as cached data only. When this data becomes invalid, new location information can be obtained from the distributed naming service [51].

Three types of reference are recognised within the proposed model:

- Connected: references given to a client and currently being actively used.
- Unconnected: references given to clients but currently unconnected to a service.
- Unknown: references yet to be handed out.

A reference that has been enabled and used for remote access is a connected reference. Relocation transparency is provided for connected references as the server object is able to send update messages directly to the client upon migration; reconnection can be established without any loss of messages or requests. This category of reference is the most common to be found within the proposed system.

Unconnected references are possible as, within a fragmented object model (or even a stub model), there is a delay between the return of the client interface fragment and activation of the reference. References are only activated upon method invocation to minimise the number of currently connected clients requiring migration updates. This time delay may coincide with a migration of the object. In this category, there must be a facility for reference updating upon a failed reference activation.

A client that has a name, but this name has not yet been resolved to a reference, has an unknown reference. This object system has no way of knowing that this future reference exists and is unable to maintain or verify the name's validity. In this case, the only requirement on the name service is to maintain validity within itself, *i.e.* its own knowledge of references must be maintained at all times.

Unconnected references and connected references are similar to unrefined handles and refined handles as introduced in the refined fragmented object model [46]. In the refined fragmented object model, handles are client interface fragments that contain information on how to contact an object. A handle can be unconnected due to the time delay between reference transmission and connection. Handles are connected immediately when the client receives them, not at a future communication point. Similarly to handles, unconnected and connected references are capable of being freely copied within the distributed system.

Figure 3 shows an example of the proposed location mechanism at work. Object A initially has an unconnected reference to object B (i), which is then connected using a client

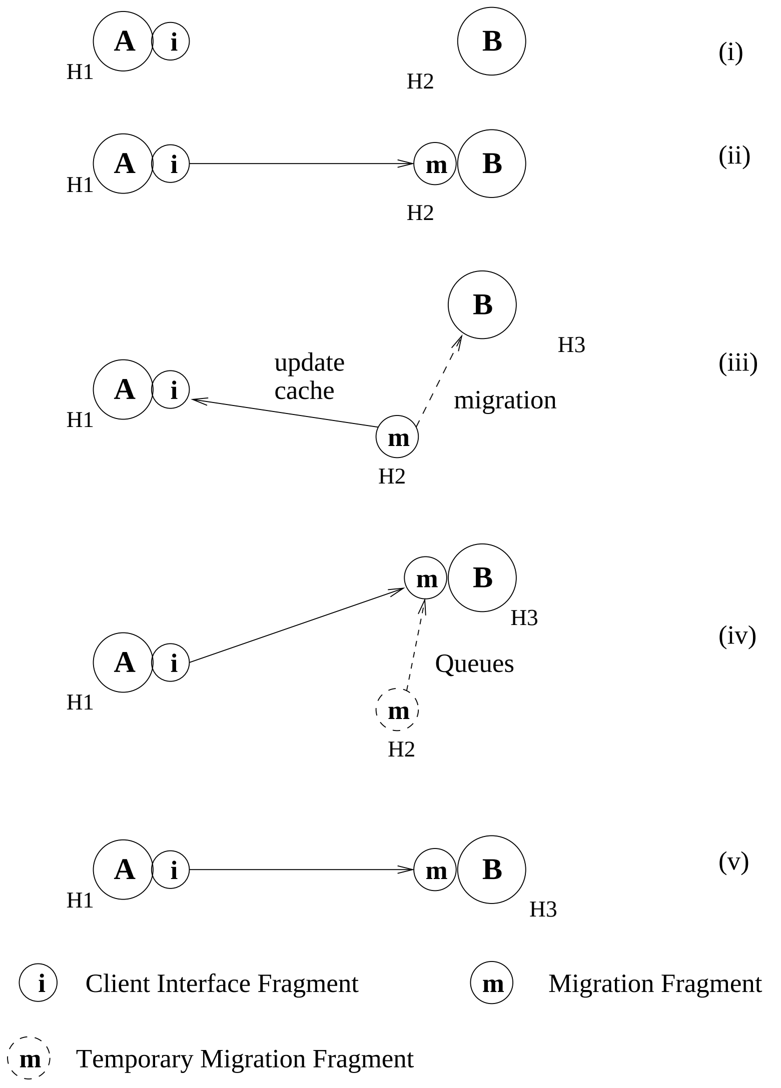


Figure 3: Location and relocation within the proposed update model.

interface fragment and a *migration fragment* (ii). The migration fragment is responsible for handling and updating connections and disappears after a completed migration. It does not act as a forwarding pointer at any stage. When object B relocates, object A's reference is updated (iii); after migration is completed and request queues have been sent to the new location, object A can reconnect and resume communication (iv). In (v) the client has a newly connected reference to the new location of B.

A migration fragment is created for each connected client. This means that direct communication and updating of connected clients can be performed, while unconnected and unknown clients are unknown to the server and must perform relocation through the name registry system. Relocation transparency is maintained by providing migration and

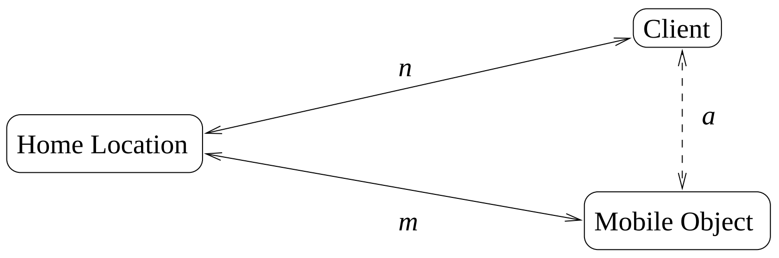


Figure 4: The trombone problem.

client interface fragments which perform any additional update tasks and location querying. No residual objects are left after object migration.

Interaction with the distributed name service is required when cached location information becomes dirty without an update from a connected server. Relocation can be performed by requesting a reference or a reference update (cached information only) for a name matching the server. A name is maintained within the client interface fragment, and consists of the object name and a unique identifier that identifies the communications channel. This identifier is used to reconnect and claim queues of migrated requests. These mechanisms are described in more detail in Chapter 6.

1.3 Comparison of the Models

One of the potential problems of the home location model is that the home location acts as a central point of failure. If the home location fails, all existing object references and any future object references become invalid. This also introduces a communications bottleneck. An additional problem is that there may be large communications overhead unnecessarily incurred if the mobile object and client are both separated by large distances (in terms of latency) from the home location, but close to each other. This is commonly known as the *trombone* problem [43, 152].

Figure 4 shows an example of the trombone problem where a client has to communicate through a mobile object's home location where the distance between the objects causes a large latency. It would be more efficient if it could communicate directly with the mobile object. The values n and m in Figure 4 represent the latencies between the home location, and the client and mobile object respectively, a represents the latency between the client and mobile object through a direct communication, where $a \ll n, m$.

In a forwarding location model, a client may have any of the forwarding pointers as their first reference to the mobile object, alleviating the bottleneck and central point of failure issues. However, for initial clients the list of forwarding pointers to visit may become arbitrarily large and potentially cyclic. A failure of a member of the chain will invalidate any references to earlier chain members. Stub-scion chains incur the same cost as the forwarding location model for the first invocation, however a shorter path can be returned or *piggy-backed* as part of an invocation response to produce a direct reference (consisting of a single SSP) for future usage.

Location management using home location and forwarding pointers are commonly used methods in the areas of mobile object, mobile process and mobile host systems [129, 130]. The problems in these methods exist due to the presence of residual objects within the object system. The use of residual objects results in out-of-date location information being used throughout the object system; additionally this information is required by clients to locate server objects. A reference that has to contact a home or forwarding location is not location independent and in some systems, not even location transparent [138].

The model proposed within this thesis makes use of a fragmented object model with migration fragments existing only for connected references. These migration fragments do not act as a chain, a single migration fragment is used as a direct reference. Migration fragments also allow updated location messages to be forwarded to the client. Unconnected references can update their cached location hints by contacting the registry system. The time between the provision of a reference and its connection is client dependent; in a system where migration is infrequent this delay will be small (minimising relocation possibilities) hence making this case uncommon.

Where relocation is common but client access is rare, the home location model does not suffer from bottleneck issues. When the scale of distribution is small, the home location model is ideal as the effect of tromboning is also removed.

In the case where relocation is uncommon but client access is frequent, the forwarding location model and, more so, the SSP chain model can be efficient as the length of the chain in each case will be small.

The proposed model is suited to the case where relocation is common and client access is frequent. References can be updated transparently for connected clients without an increasing chain of forwarding locations. Bottleneck issues are reduced due to the distributed nature of the registry system and direct referencing. All models benefit from a smaller scale of distribution due to the reduced latency costs and reduction of any potential trombone effect.

1.4 The DISCWorld Metacomputing Environment

The proposed model of location and relocation transparency based on top of a distributed registry system is implemented as part of the Distributed Information Systems Control World (DISCWorld) metacomputing environment [78, 82, 104].

The DISCWorld metacomputing environment provides middleware support for distributed systems. It supports web-based client access to legacy and specialised services running within the DISCWorld system. DISCWorld supports dynamic reconfiguration and adaption in response to data access requirements and processing load within a data-intensive high-performance system. Additional services that DISCWorld provides are meta-data resource discovery, data transport, process scheduling and monitoring services.

Several high-performance legacy applications have been developed to support Geographical Information Systems (GIS) within the DISCWorld framework [37, 38, 103]. These services have been integrated into the DISCWorld metacomputing environment and provide a mixture of mobile and host dependent code; and code implemented for high-performance and low-performance systems.

1.4.1 The DISCWorld ORB System

The distributed registry system is based on an ORB model, and acts as a name server and a framework for linking additional services into the DISCWorld ORB system [51, 104]. The DISCWorld ORB system is designed for native mobility support; this is implemented on top of the ORB system as an additional service.

An ORB model defines a model for client/server communications utilising an independent third party. Access to this third party may be transparent or it may be direct depending on the system model or even on the types of communications and reference access used within the same model. Examples of well known ORB models are the Common Object Request Broker Architecture (CORBA) [136, 138] and Java RMI [70, 167]. These are commonly identified examples of ORB systems, however many other distributed systems fit the ORB model, including Globus [54, 57] and Globe [177–179].

Essentially a distributed object system based on an ORB model is one that uses a third party object to provide references and additionally manage name binding and name resolution to some extent within client/server communication. A server will register its service and potentially bind a name to this service (alternatively a name can be provided by the naming system itself) through some facility provided by the ORB. This could entail binding to an external ORB service [136] or accessing internal library code [39, 167]. A client will request a named service from the ORB and obtain a reference, generally in the form of a stub, to abstract over remote communication requirements.

The ORB provides a mapping between a name and the service object and is able to provide stub code or a method to obtain stub code as provided. Figure 5 shows the operation of typical client/server communication within an ORB model. Initially (i), a server object registers its service or name with the ORB. When a client requests name resolution by the ORB, a stub is returned to the client (ii). This stub can then be used to remotely invoke methods on the server by communicating with the server's skeleton object (iii).

An ORB model is useful for dynamic distributed object systems as it acts as a well known contact point through which clients and servers can communicate. A publicised port is commonly used as the ORB contact point. If communications libraries are provided to

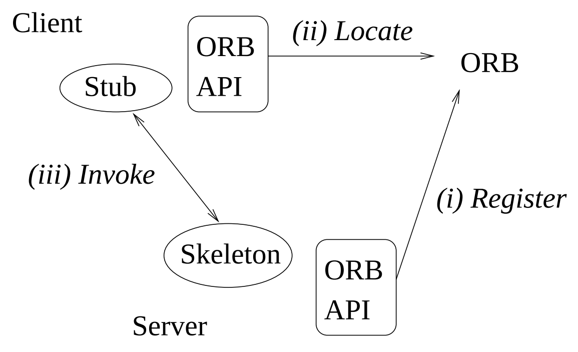


Figure 5: Stub-based client/server communication within an ORB system.

the client and server objects, this enables an ORB system to be accessed without explicit remote communications required by the client and server code.

ORBs are often used as well known access points for additional services provided as part of the distributed object system. Some commonly found services include security services [136], trading services [17, 137] and contextual naming systems [135]. Mobility services have also been proposed [128], while ORB-based systems with inbuilt mobility have also been proposed [131].

The DISCWorld ORB system provides mechanisms for clients within the DISCWorld system to register services, obtain references to services and perform optimised communications. To support scalable wide-area communications, a communications construct has been developed which utilises the inbuilt support for mobility and transparent reference updating (see Section 6.6). The DISCWorld ORB system provides additional service support for object migration, replication and cloning.

The DISCWorld ORB system has a hierarchical directory structure with adaptive components and a distributed nature. Different models for updating these directories are used dependent on the level of the registry within the hierarchy and the expected latency within the system. The DISCWorld ORB's naming model is consistent with the extended naming model detailed in Chapter 5 and provides support for service relocation, name aliasing, name extension and dynamic reassignment of names to objects. This naming model avoids polling of services and provides a consistent communications scheme to support the distributed directory service.

1.4.2 Distribution and Object Models

The distributed object model used within this thesis an example of a fragmented object model. A client interface fragment or stub is used to separate the communications from the application object. Client/server communications then occur through the stub and the group interface fragment. Additional fragments are used to perform the low level communications within the system and fragments are required to interface to the naming system.

Objects are accessible through predefined, typed methods either locally through direct invocation or remotely through the stub interface. Multiple stub interfaces can be accessed by the system to support multiple protocols and multiple client types.

Objects that can be accessed or referenced remotely by another object must be named. Local access need not be named although this is not prevented. Additionally objects passed or returned in a remote invocation are passed by reference if named, but by value otherwise. Naming is performed on a per-object basis so that multiple instantiations of the same class or code may have different names. An object is named if it is registered with the ORB system or has been exported as part of a remote communication; objects that are only accessed locally are not named. Primitive types that do not exist as objects can not be named.

Names are constructed using a contextual mechanism that supports three contexts. Objects are accessed through their code defined names (such as variable names) within the object's private address space. Local names can be used within the local name registry system that consists of a small collection of nodes involved in a computation; global names can be used universally between name registry systems. An object may have multiple names (global and local) where the local names can only be used within their valid context. User defined contexts beyond these facilities can not be defined.

1.5 Contributions

This thesis discusses models for location and relocation transparency used within existing mobile and distributed object systems. A new model, based on the combination of an explicit update model and an ORB-based distributed directory system, is introduced. Location transparent and independent naming to support this model is provided through a global namespace managed by the ORB-based distributed directory system.

This thesis describes the development of a formal naming system classification model based on existing work in the area by Bayerdorffer [15, 16] and Bowman *et al* [26]. Attention is also payed to other naming models preceding Bayerdorffer and Bowmans' work [50, 63, 89, 92, 158, 159]. This naming model is used to define a separation between the naming system and the remaining parts of the system in which it will be used. The naming models defined by Bayerdorffer and Bowman *et al* are used to classify existing mobile and distributed object systems. This classification shows that these models do not correctly or completely classify these kinds of object systems. Extensions and refinements to these models are proposed that enable a complete and correct classification of existing mobile and distributed object systems, specifically with a need for transparency.

Bowman *et al* define a formal model for defining name resolution systems. This formalisation is extended to formally define first Bayerdorffer's name binding model and then the extended naming models. The definition of a formal classification model enables the development of a generic support framework for naming systems.

An implementation of the proposed relocation transparency model within a distributed object system with support for mobility is presented. The DISCWorld ORB system is introduced as a motivating framework for the transparency and naming models. The DISCWorld ORB system provides a global namespace managed by a naming system. The implementation of this naming system is separated from the remaining implementation and is based on, and classifiable by, the extended naming system classification model.

The DISCWorld ORB system supports an Application Programming Interface (API) for client/server access to the registry system. This API allows transparent access to the distributed directory service without requiring client/server knowledge of directory location or structure. The DISCWorld ORB APIs provide support for object migration, cloning and replication. This system provides an API for dynamically constructing mobile services to perform batched remote invocations. This construct enables a sequence of connected remote invocations to be programmed as an itinerary. The mobile object is then responsible for colocating with the required service objects to perform more efficient local invocation.

This thesis describes an implementation of a prototype of the DISCWorld ORB system which supports distributed and scalable object location and relocation. Experimentation with the DISCWorld ORB system has shown that it provides better performance than existing models for relocation transparency in terms of cost performance and scalability. The DISCWorld ORB system also provides support for fault resilience and removes central points of failure.

This thesis makes three main contributions. The first contribution is the development of a formal naming model that is suitable for classifying existing mobile and distributed object systems. The formal definition of this naming model enables support for generic naming systems to be separated from the development of the remainder of the object system. Separation in this way enables the naming model that is used within an object system to be changed and updated according to requirements of the object system.

The second contribution of this thesis is the development of a model to support location and relocation transparency. This model provides relocation transparency using location transparent and independent references and has no central point of failure or bottleneck issues.

The third contribution of this thesis is the development of a prototype distributed ORB system, the DISCWorld ORB system, that supports the proposed generic naming model and the model for relocation transparency. The DISCWorld ORB system is implemented as a distributed, hierarchically structured, ORB system that is capable of fault resilience and adaption. Experimentation with the DISCWorld ORB system shows that the proposed model for relocation transparency provides better location and relocation performance and is scalable in terms of the number of nodes within the system, the number of names within the global namespace, and the frequency of client requests.

1.6 Thesis Structure

This thesis is divided into three logical parts. The first part introduces the problem and defines the problem area through a literature review. This introductory chapter outlines the area within which the rest of the thesis falls and outlines the main contributions of the work. This chapter motivates mobility in distributed object systems and discusses additionally the need for naming in distributed object systems for client/server communication. It outlines how naming can provide the facility for mobility and transparent locality within a distributed object system. A system is outlined, based on an ORB-style system where an expressive naming model is used to provide location transparent and independent mobility. The benefits of this system are outlined and examined. Chapter 2 examines existing mobile and distributed systems, specifically looking at each system's support for location and relocation transparency and distributed communication.

The second part of this thesis discusses the issue of naming. Chapter 3 presents problems in the area of naming within mobile and distributed object systems. Naming models used to classify naming systems, including the models defined by Bayerdorffer and Bowman *et al*, are presented in both an informal and formal manner. In Chapter 4, these naming models are then used to classify the naming systems of existing mobile and distributed object systems. A taxonomy of the classified naming schemes is presented. This classification is used to highlight deficiencies in the current models. Chapter 5 presents motivation for extending the naming models to provide characteristics suitable for distributed object systems with mobility and distributed systems constructed around an ORB model. It presents the extended model and uses it to form a reclassification of the systems described in Chapter 4. This extended model is then used to define a naming system suitable for location and relocation transparency in a distributed object system with mobility support. An expressive naming system is presented and it is shown how this system can support transparent object mobility by providing a state dependent mixture of location dependent and location independent mappings.

The third and final part of this thesis presents the example system that has been developed as part of this work and shows how this system is used to support the naming models and relocation model developed throughout parts one and two of this thesis. Chapter 6 outlines the system that has been developed to support mobility of objects within the DISCWorld Metacomputing system. It presents the naming model and its classification suitable for use with a mobile object system. Chapter 6 also discusses how the naming model can be extended to a distributed directory system with partially contextual names.

Chapter 7 presents a discussion of the implementation of this system and how tasks such as distributed aliasing were achieved. Chapter 8 presents an evaluation of the extended naming model and its effectiveness within a distributed ORB system supporting mobility. A comparison between the performance (with respect to both relocation cost, location cost and communication cost) using the proposed model for transparency and existing models is presented.

Chapter 9 presents conclusions from this work and outlines future work that is still to be done in the areas of relocation transparency and naming.

Appendices present more detailed descriptions of the policy specification and communications protocols that are used within the DISCWorld ORB system. APIs for client and server access are also provided.

Chapter 2

Relocation Transparency in Existing Systems

Mobility has been implemented in the forms of process mobility, object mobility and host mobility. The same essential issues occur in each of these areas: how do the objects move, how are they located, how are they tracked, and how are they identified or named.

Process mobility is concerned with the relocation of active processes between nodes within a distributed or parallel system. One of the most important issues in process mobility is that of state transferal. The system needs to be able to access the execution state of the process in order to migrate it seamlessly. This requirement often leads to process migration systems being encapsulated within purpose built operating systems.

Object mobility operates at a higher level than process mobility and may also have a requirement for state transferal. Some object systems use techniques such as checkpointing or manual state recording as an alternative to direct state capture. These alternative techniques allow object mobility to be implemented above an operating system in an abstract manner.

Host mobility is concerned with mobile computers or hardware that can be reconnected at, as yet, undetermined points. State transferal is not an issue here as the mobile computer itself handles all of its own details. The important issues in host mobility are the locating and tracking of hosts. Several specifications and systems for tracking and providing directory services for host mobility have been proposed [6, 152]. These systems are generally based on simple home location or forwarding location models.

Mobility can be either autonomous or controlled by the surrounding system. Autonomous systems (agent systems) often do not allow other agents or objects within the system to have references to the agent, or if external references are permitted, relocation transparency is not provided. Consequently, many autonomous object systems do not provide a means for uniquely identifying their objects or for relocating them.

Mobility can also be either weak or strong. Weak mobility is defined as mobility of data state, strong mobility is defined as mobility of the execution state as well as the data state [41]. A strongly mobile system is capable of migrating an object or process at any point as it can capture all of the object's state. Strong migration requires full relection of the system's state. A weakly mobile system is only capable of migrating an object at certain points where the data state has been confirmed. A weakly mobile system may depend on a checkpointing system or on predefined methods for requesting data state commitment.

Weak mobility places restrictions and additional requirements on the objects within the system that are not necessary within a strongly mobile system. However, weak mobility can be implemented in a simpler, platform neutral way with no requirement for methods to access the execution state of the object. For an interpreted implementation language, often the case for mobile object systems as it can provide additional security abstractions [115], this is an important requirement.

Cugola [41] outlines three categories of distributed systems with mobility support:

Remote Evaluation: An object can invoke services within another object in the distributed system by providing code as well as parameters, which define how the service will be performed [165].

Mobile Agent: A single object or executing unit that moves autonomously from host to host continuing execution at each host seamlessly.

Code on Demand: A client can request and download code on demand to perform some task. This code may be updated at the download site to reflect version changes.

The remote evaluation model is used in systems where an object is dynamically bound and migrated to a remote host to perform some action. The migration is not autonomous and can be performed at a process or an object level.

The mobile agent model is a generalisation of the autonomous agent model. A mobile agent is capable of migrating through some programmed itinerary or it can design its own itinerary, dependent on dynamic properties. Examples of this category include [72, 113, 115, 156]. Mobile agents are inherently not migration transparent due to their autonomous nature.

The code-on-demand model is used in systems where code repositories or libraries are essential to the system's execution. These libraries can be dynamically updated, thereby producing adaptive systems. Code objects from these libraries can be downloaded in order to update the current code listing of a stationary object. Such systems, for example Java [69], work by having clients request code objects from remote sites (such as web sites) that can be updated and changed by some server management system [125].

Mobile object systems that allow external references to their objects often require a directory or registry system that supports object location. These directory services can be used to support relocation of references when objects migrate but often this is performed in a nontransparent manner [14, 71, 115].

Distributed object systems that support mobility have much in common with mobile object systems, mobile host systems, mobile process systems and also distributed directory systems for distributed shared data. Many of these systems work by using a naming system to name either its objects or its routes to the objects. By examining the naming systems and transparency support in both mobile and distributed object systems, it is expected that this discussion will outline points of common interest and areas where mobile object systems can benefit from naming support of a widely distributed nature.

This chapter presents a discussion of the support for relocation and relocation transparency within existing mobile and distributed object systems. Process migration, host migration and autonomous agent systems are also examined. These systems are also examined with regard to their support for location independence, mobility and distributed communication. A comparison of the mechanisms used and each mechanism's advantages and disadvantages is provided for each system, as is a discussion of existing naming systems for each system.

2.1 Mobile Process Systems

Process migration precedes object migration as an area of research interest and has led to many of the ideas and techniques that are found in today's mobile object systems [129]. Systems that support process migration are strongly mobile systems as they must support full state transfer of the migrating process. Process migration systems are often dependent on a choice of platform or operating system due to this constraint.

Process migration is of interest to this discussion because the transparent relocation of processes is a requirement of such a system. Process migration requires that all references be valid at all times, including low level references maintained within processes and the operating system. This can not be concealed through the use of proxy code or exception handling techniques as found in mobile object systems. Process migration systems are often found in machine clusters or operating systems which present a consistent and unified view. The scale of these systems is generally small and is not at the scale of World Wide Web (WWW) based systems. This characteristic reduces the effect of potential tromboning or chaining of forwarding pointers, making these more suitable models for relocation transparency.

Process migration systems generally have common naming and location characteristics. Each process is given an identifier (ID) which is either globally unique, or is locally unique and is globally interpreted through an adaption mechanism. A process name consists of its ID and the ID of the processor on which it is executing. The processor ID is changed depending on the process object's current location. Process names are neither location independent or transparent. Process migration systems are typically migration transparent.

2.1.1 Charlotte

Charlotte [3,4] is a distributed operating system that provides process migration. Charlotte is designed to operate over a multicomputer of 20 VAX-11/750 computers using message-based communication. Charlotte provides a separation of migration policy from migration techniques. The policies for process migration (*i.e.* using a load balancing or load sharing scheme) for a particular Charlotte system can be set at system instantiation and changed dynamically throughout system lifetime. The mechanisms that actually provide migration are part of the system kernel and hence can not be altered.

Charlotte provides a unique implementation of a forwarding scheme to provide relocation transparency. References between processes in Charlotte are connected by a duplex channel, which means that both processes involved in the communication know about the other. This allows a message with the new location of the object to be sent by the migrating process to the referencing process. Charlotte also leaves a residual forwarding pointer behind so that any further accesses to the process due to garbled messages can still be forwarded to the migrated process.

When a Charlotte process migrates it performs an atomic migration. If the migration is successful then update messages are sent and queued requests are forwarded to the migrated code. If the migration is not successful then no change is apparent to referencing objects. The state of a reference when an attempted migration fails, or moves on before reconnection could be established, is unspecified by the implementors of Charlotte [4].

2.1.2 DEMOS/MP

DEMOS/MP [149] is a distributed operating system that supports transparent process migration. DEMOS/MP provides a location transparent communications interface that provides some of the mechanisms required for transparent migration.

Communications are sent using *links*, which are protected and transparent global addresses. A process may have multiple links to it present within the system; these links are used to communicate with the process via a message-passing communications model. A link ID consists of a combination of a unique processor ID and process ID which is a system wide unique identifier. During process migration it is the first component of the link ID that must be updated.

DEMOS/MP makes a distinction between the update requirements for user and system processes [149]. User processes are thought to have external references resident in system processes only; these references are generally short lived and hence no updating is required.

System processes, on the other hand, may have external references in user processes that may be long lived and these do require updating.

DEMOS/MP uses a lazy forwarding pointer mechanism to update its links. When a process migrates, it leaves behind a special process in its place that acts as a forwarding pointer. When a process attempts to access a migrated process, it instead accesses the forwarding pointer process. The DEMOS/MP kernel catches this access and updates the links in the referencing process's link table to reference the new, or last known, location. Communication can then be performed transparently through the updated link. Any new links that are created will also be caught using this mechanism. If a forwarding pointer process crashes, normal process recovery mechanisms used by DEMOS/MP can be used to recover the forwarding pointer.

Links are only updated as is necessary, hence it is possible that links will exist indefinitely with incorrect information. To deal with this DEMOS/MP never reclaims its forwarding pointer processes. In Powell *et al* [149] mechanisms are outlined for reclaiming forwarding processes, however it has not been found to be a requirement in DEMOS/MP.

2.1.3 MOSIX

MOSIX [12, 13] is a multicomputer operating system that presents a single image of a cluster of homogeneous UNIX-based nodes. MOSIX provides a separation between location dependent and location independent components of the kernel, thereby allowing processes to have an abstract view of the processor they are currently executing on. MOSIX provides a unified view of the cluster with a single view of the distributed file system.

MOSIX provides transparent process migration [11] and transparent access to remote objects and resources in two ways. One is through a home location or *deputy process* and the other is through a *linker* that transforms local communications and requests for resources to remote objects through transparent network access.

Process migration is only allowed between processors with the same instruction set. Processes in MOSIX can not be migrated while they are actively executing an RPC until the process reaches an appropriate point. An appropriate point for process migration can be the reaching of an idle state or the completion of a method invocation.

In MOSIX, processes are split into a deputy process and a *body* process. The deputy process resides at the node where the process was created and acts as a home location, the body process (the actual process) can then be freely migrated around the nodes in a cluster.

References to the body process are directed to the deputy process, the deputy process is updated upon process migration so that it contains the actual location of the body process. A special, optimised communications channel connects the deputy process to its body process, which allows messages and RPCs to be forwarded in an efficient manner.

This channel acts as a single point of failure, however, which can temporarily delay all communications between referencing objects and the mobile process.

2.1.4 Sprite

Sprite [49,142] is a networked operating system that provides process migration without the use of residual code. Sprite uses a home location model for process migration which provides an interesting side effect for processes. Each process has a designated home machine that is the machine that the process was created on. Regardless of its current location, the process always appears to be resident on the home machine, even down to operating system programs such as *ps*, which produces listings as if all migrated processes were local. This allows migration to be transparent to the user of the operating system and the migrated process.

Similar to MOSIX, processes in Sprite can not be migrated while they are actively executing a RPC until the process reaches an appropriate point.

Sprite provides four mechanisms for providing transparency; the first is location transparency through a shared name space, allowing system calls to be location independent. The second is a complete transfer of state when a process is migrated, leaving no residual objects behind. The third is the use of a home machine that can perform message forwarding to migrated processes, and the use of caching to produce direct forwarding if possible. The home machine model can be used to separate execution of calls, some of which may be executed on the home machine and some of which have to be executed on the current machine. The fourth mechanism for providing transparency is cooperation between the home machine and the current machine of the migrated process to ensure that any created or forked processes have consistent process ID's regardless of which machine they were created on.

2.1.5 V-System

The V-System [124] is a distributed operating system that provides process migration and transparent remote execution facilities with the aim of taking advantage of idle processors within a cluster. The abstractions and migration facilities implemented in V have been implemented almost completely outside of the kernel, unlike systems such as MOSIX and DEMOS/MP.

V provides an abstraction over locality by providing logical host IDs. A logical host ID is used to define a single node, with a node potentially having multiple logical host IDs. V does not require the use of residual code or a home location. Relocation transparency is provided through a broadcast mechanism within the cluster to find the new location of the migrated process. A broadcast mechanism can cause network saturation in a object system with frequent migration and client access.

When a V process migrates, it is suspended pending the migration and any requests are queued and a *reply pending* message sent back to the referencing object. When the process has completed its migration, all queued messages are discarded and the referencing objects are required to re-send their requests to the new location. If the client process' reference is a local reference, then the message is re-mapped to a remote invocation; if the client process' reference is a remote reference, then the remote invocation has its destination updated to be the logical host ID. If a reference fails, *i.e.* the logical host ID does not respond, then the reference is broadcast over the cluster to attempt to find the new, correct logical host ID.

2.2 Mobile Object Systems

This section examines existing mobile object systems including autonomous systems and mobile code systems. Each system is examined relative to its support for location and relocation transparency and its particular form of migration. The analysis includes the system's inherent support for distribution (*i.e.* whether communication can be performed on a remote as well as a local basis).

2.2.1 Ajents

Ajents [95] is a Java-based mobile object system that utilises Java RMI [55, 167] for its communications mechanisms. An Ajent is a mobile agent that can be relocated to respond to load balancing requests but is not autonomous.

Ajents supports asynchronous Java RMI and load balancing through dynamic migration of remote objects. It relies on the code loading capabilities and naming models of Java RMI to perform its underlying communications and reference management. Objects can access and bind to Ajent objects through traditional RMI methods. Since Ajents relies on Java RMI for the underlying implementation it is limited by the scalability of Java RMI as a distributed communication mechanism; no distributed directory structure is proposed to increase its scalability.

Ajents introduces the concept of an Ajents Server that exists on each host participating in the distributed system. Each Ajents Server is responsible for monitoring and registering mobile objects within its system and for performing mobility and relocation tasks. Ajents Servers support Ajent migration by suspending the mobile object in response to a migration request and performing any migration tasks on the Ajent's behalf. The Ajents systems provides weak migration, restricting it to objects that are currently idle.

Objects to be migrated are first suspended and then checkpointed by the Ajents Server at the first available opportunity (idle state) and then the object and the checkpointed state are migrated. After migration is completed, the Ajents Server contains a forwarding reference to the new location. Upon the next access to the mobile object, and after following the chain of forwarding pointers, an exception is raised that contains the new location of the mobile object. This exception is caught and handled by the underlying Ajents framework; causing the client's reference to be updated with the new location and resubmission of the request. This mechanism provides relocation transparency.

2.2.2 d'Agents or Agent Tcl

d'Agents [71, 72, 110] (formally known as Agent Tcl) is a Tcl-based autonomous agent scripting language that supports strong object migration. Objects in d'Agents can be programmed with an itinerary and may migrate at any point. d'Agents has been extended to support other languages including Java, Scheme and C++.

Objects within d'Agents consist of autonomous agents and locations where agents can reside. Agents can migrate between locations within the agent system. Each location must contain a d'Agents server that provides the code required to accept agents. There may be multiple agent servers on any one host. Objects within d'Agents refer to each other using symbolic names which can be used to open communications channels on a local or a remote basis.

Each object within a d'Agents system contains a list of packages that it uses; these packages are required by the system to create or host the object. Code libraries are located at various points within the system. These libraries can be contacted by a d'Agents server to facilitate downloading of required packages. Code is loaded through a hierarchy of libraries until potentially it attempts to load the library from the network. This is similar to the classloading concept in Java, where defined classloaders can load and name class objects within their scope, potentially from network locations with the same degree of security.

A d'Agents server maintains a list of those agents currently residing with it. It also maintains a nonmobile *yellow-pages* style list of services. A d'Agent can register its service with the yellow-pages, matching a service name to a semantic description. An object can register multiple service types that it is able to support. A yellow-pages listing contains a list of services and also a list of other yellow-pages listings that are known to the current d'Agents server. Remote yellow-pages can not be contacted directly but must be visited and queried in turn, taking advantage of the inherent mobility within the system.

A object can obtain a reference to a d'Agents object using one of three methods:

- direct reference through a hardcoded agent name plus location reference,
- querying of an agent server to list its current servers, or
- querying of a yellow-pages listing to check the current registered servers and the location of other yellow-pages listings.

Communication channels (remote references) require the specification of a location, meaning that d'Agents does not support location or relocation transparency. Once an object has migrated, the channel must be reopened with the new location. d'Agents does not provide any way of explicitly informing a referencing object of a migration or location change, depending on the yellow-pages listings for object tracking.

2.2.3 Emerald

Emerald [21,91,102] is an early mobile object system designed to support inherently mobile objects within a distributed shared memory object system. It has been said that Emerald was the mobile object system that lead most directly to mobile agent systems [185].

Emerald is designed to support both fine-grained data objects and coarse-grained process objects. Emerald is also designed to experiment with mobility in a distributed system [102] and hence has extensive support for relocation and interprocess communication. Emerald provides system level support for migration and also provides language level support. A programmer can specify that some of its objects be local, and some remote, thereby allowing specific objects to migrate to other nodes. Factors such as invocation and parameter passing are all linked to native migration support rather than being implemented on top of migration support as in a typical process migration system [3,12,149].

Each object has four components:

- a unique network-wide name,
- a representation or data state local to the object (data and references),
- a set of operations that can be invoked, and
- optionally, a process.

Objects that contain a process are active (coarse-grained) objects, whereas objects without a process are passive data (fine-grained) objects. The same construction mechanism is used for both fine-grained and coarse-grained objects.

Objects can be defined as either global or local[2]. Global objects may be migrated anywhere in the distributed system and may also be referenced anywhere; local objects are contained within another object and may not have external references. Local objects can not migrate independently of their container. Emerald objects are defined using a single definition mechanism, regardless of their locality. The Emerald compiler deduces certain characteristics about the object (such as whether it is globally shared, or is purely local) and then produces the correct mobility support for it.

[2]An additional form is *direct* which is mainly used for primitive types and does not have implications for mobility.

Each object is given a globally unique object ID. This ID can be resolved to a reference, which may take the form of a local address or be the address of a forwarding pointer. Each node within the cluster has an *access table* mapping a object ID to an object descriptor. The access table on a node contains an entry for all local objects and for all remote objects that have a resident forwarding pointer.

Following a forwarding pointer will cause an attempted resolution on the remote node that the forwarding pointer references. This procedure repeats recursively until either the object is located or the resolution fails. If the name can not be resolved, a broadcast protocol is used in an attempt to locate the object. These mechanisms together provide relocation transparency.

Emerald has inbuilt support for interprocess communication using RPC-style semantics. This, and its support for broadcasting and resource location over a cluster of nodes, provides support for distributed computation on a medium scale. Emerald is not designed to run on wide-area networks but rather on small distributed clusters with an expected number of nodes of less than 100 [102].

2.2.4 MOA

MOA [131] is a mobile object and agent system developed to support migration and communication between distributed objects. MOA supports an integrated naming scheme and resource location. MOA objects are autonomous objects that migrate between *places* within Agent Environments (AEs). An object is able to execute within a place.

An agent is able to request relocation, and communication, between itself and another agent. Agents communicate by specifying another agent's name. Agents are identified by a combination of the name of the environment in which they are first created, and a unique identifier. Due to the location dependence of agent names, communication between agents does not require explicit destination location specification as is often required in mobile agent systems.

MOA outlines a framework for migration in a distributed system integrated using distributed name servers and agent servers. MOA agents can migrate to any node within the system and make use of the name service at any stage or location. MOA provides multiple mechanisms to support transparent relocation and also mechanisms for resource discovery. However, these mechanisms rely on the assumption that it is possible to find out the location where the agent was created in order to obtain the agent's name, and hence, that names are neither location independent or transparent.

Agent location is performed by one of several name servers present within the agent system. A name server provides mechanisms to look up a name, register a name and remove a name. Name servers are ordered in a hierarchical manner and provide search mechanisms for the agent names provided by the system. If the name server is unable to resolve a name, then the agent's home address can be located for a reference. The home address is always identifiable as it can be derived from the name of the agent.

MOA supports four models of relocation: the home location model, the forwarding location model, a central registry model, and a search-based model (using methods such as itinerary searching or broadcast queries). This allows different policies to be used for different agents within the one system. The policy used depends on how the agent is to be used. For example, an agent that does not move frequently but moves large distances (in terms of latency) away from its creation node is suitable for a forwarding location model. An agent that moves frequently but within close proximity to a potential home location or registry is suitable for a home location model. A search mechanism may only be suitable for agents that have a predefined path of migration so that all possible nodes are known in advance.

When an agent migrates to another host it leaves behind a *location* object that contains either the current location of the object or information about the correct strategy that can be used to relocate the object. This information details the type of strategy used and information required to deploy the appropriate strategy, such as the address of a home location. Location objects are cached at each host that the agent visits so that, given frequent migration, there is a reasonable chance of finding a given location object (regardless of its age) within the system. If the object can not be found then the request is forwarded to the home address of the agent.

The mechanism used to support relocation transparency in MOA relies on the system using names that are neither location independent or transparent. If an object was required to change its mechanism of relocation, it would have no means of efficiently updating all of the location objects that had been cached in the system.

2.2.5 Obliq

Obliq [29,30,94] is an object-based language developed by Luca Cardelli. Obliq is a lexically scoped, dynamically typed and interpreted language which is not natively mobile, however objects can be relocated through explicit redirections. Objects can be transmitted across the network through parameter passing in remote method invocation, providing a controlled means of migration. Objects can also be cloned in a transmission; this form of transmission does not actually perform object migration but causes the reproduction of objects at remote locations.

Execution Units (EUs) in Obliq create objects which are stored within the EU's local data space. When an object moves, *i.e.* is passed from one EU to another, the local references are transparently changed to network references. Accesses through references to network objects are transparently changed to callbacks to the EU that holds the appropriate object. A consequence of this is that Obliq objects can roam freely over a network while maintaining network connections in the form of channels. In this way Obliq supports completely transparent network and mobile programming.

Obliq supports a name server that performs matching between object names (in a textual form) and network references. A single name server is used within an Obliq system which resides at a well known location. This forms a centralised directory system that encourages communications bottlenecks in a similar fashion to the home location model.

Relocation is performed through a combination of a forwarding location mechanism and updates to the name server. Each object that has a remote reference to an object that is migrated or cloned will have its reference redirected to an alias. This alias contains information on how to relocate the object. Objects that wish to obtain references to the migrated object for the first time can obtain a direct network reference through the name server.

2.2.6 Sumatra

Sumatra [1, 151] is a Java-based mobile object system that supports resource awareness. Sumatra supports the ability to monitor the levels and quality of resources at a location, the ability to react to changes in resource levels, and the ability to control (as far as possible) the way that resources are used by them. Sumatra is primarily designed for systems that require semi-autonomous agents to react to, and inform external agents about, resource changes within a network. Sumatra is an interesting mobile object system as it supports a form of object relocation without relocation transparency.

Although Sumatra is Java-based, it is still a strongly mobile object system as it does support full state transfer. Sumatra provides native C code that is able to access the state of the mobile object; this native code forms part of the underlying agent server code. Hence, the agent code is still pure Java and can be transferred regardless of the underlying platform. Sumatra objects can only migrate as part of a migrating object group and migrate to a named location.

Sumatra adds four new concepts to the Java language:

- An *object group* is a dynamically created group of objects which are treated as a unit for mobility purposes.

- An *execution engine* is an execution point or interpreter on a host.

- *Resource monitoring* which allows Sumatra objects to query and request resources from external agents

- *Asynchronous events* signify high priority changes in resource levels to objects residing within an execution engine.

An object group has the ability to control an enclosed object's life-time, and the ability to remain in existence after the creating object has terminated or relocated. An object belongs to all of the object groups that contain it, ordered in a hierarchical fashion.

Object location is performed through the specification of the required object's ID or a local symbolic name, and the name of the execution engine where the object is resident. Names for execution engines are a combination of the IP address of the node and a port number where the engine can be accessed. These names are neither location independent or transparent. This mechanism allows a Sumatra object to have external references and hence support distributed communication between objects using a form of remote method invocation. Sumatra provides no mechanism for resource discovery.

If a referenced object migrates, it leaves a forwarding pointer behind that points to its new location. This forwarding pointer is not accessed as a chain, but is used to pass the new location information back to the referencing object. When a referencing object attempts to invoke methods on a migrated object, an exception is thrown by the forwarding pointer. This exception contains the new location, allowing the referencing object to reconnect its reference to the mobile object at its new location. If at that time, the mobile object has migrated again, another exception will be thrown, and so on, until the object is located.

The relocation mechanism used in Sumatra is similar to a forwarding location model but is not transparent. However, the mechanism used is more efficient than a forwarding location model as it does not require communications to be forwarded through a chain of pointers. The efficiency shown is similar to that of an optimised forwarding pointer mechanism as shown in a weak reference in the stub-scion mechanism (see Figure 2 on page 7). In a stub-scion mechanism, the forwarding chain is used for the first invocation and then a direct reference is returned with the invocation result. In the method used by Sumatra, it is in the stage of reference resolution that the direct reference is returned, permitting all invocations to proceed by direct communication.

2.3 Mobile Computer Systems

Mobile computer systems face many of the same problems as mobile object or process systems with the additional burden of a well-defined naming protocol for hosts (using IP [148]) that is neither location transparent nor location independent, and is defined with the assumption that objects are stationary.

Mobile computer systems have additional issues to contend with in that they must cope with consistency problems. If a computer is removed from its network, then it is unable to verify such things as access permissions or file consistency. Several systems have been developed to support reintegration of file systems with mobile components, including [90, 100, 107, 173]. Jing *et al* present an extensive overview of work in this area in [99].

What is of greater interest is the development of Mobile IP [93], which has been designed to extend the location dependent naming scheme (using IP) to support location transparent addressing for mobile computers. Directory systems to support transparent relocation of hosts within a specified network have also been developed [5,6].

2.3.1 Mobile IP

Mobile IP [93,146] is a proposed standard for transparent IP naming of mobile computers. IP is unsuitable for naming of mobile computers in that it represents a location specific name. IP addresses can be used to define the location of a host and a potential route to the host. This is unsuitable in a mobile host system.

Mobile IP uses a home-location-based mechanism for relocation where each mobile computer has a home location specified by a Mobile IP address. This IP address forms part of a virtual name space which represents virtual subnets for groupings of mobile IP addresses; the mobile IP does not indicate a structure or physical routing to the host. All mobile IPs have addresses within the mobile subnet.

Each mobile host uses two IP addresses. One is for the home location and is static and the other is termed the *care-of* address and is used to represent the mobile host's current location. Use of a home address allows the mobile host to be always contactable in its home network using standard contact mechanisms. Packets that the home address receives are redirected (changing the destination IP address) to the mobile host's current care-of address. The mobile host is responsible for maintaining the home address's knowledge of the IP address of the care-of address.

Mobile hosts can obtain a care-of address using discovery mechanisms built into the network. A mobile host must make sure that the new address is free for use. This mechanism uses mobile agents that advertise care-of addresses which are available; this mechanism is based on top of existing protocols for router advertisement. Advertised care-of addresses may be taken freely by the mobile host.

Mobile IP has facilities to cope with home address failure. If a mobile host can not contact its home location, it can send a broadcast registration update to its home network, which is received by other accessible home addresses within the network. When these home locations respond with a rejection, the mobile host can use these rejections to select a new home address to which to send a registration request.

2.3.2 Regional Directories

Regional directories [5, 6] is a distributed directory system for managing mobile hosts. Regional directories use a partial information scheme where directories exist within a hierarchy organised by regions. Within a region, directory information is fully replicated. This form of directory structure limits the required broadcasting of updates in a replicated

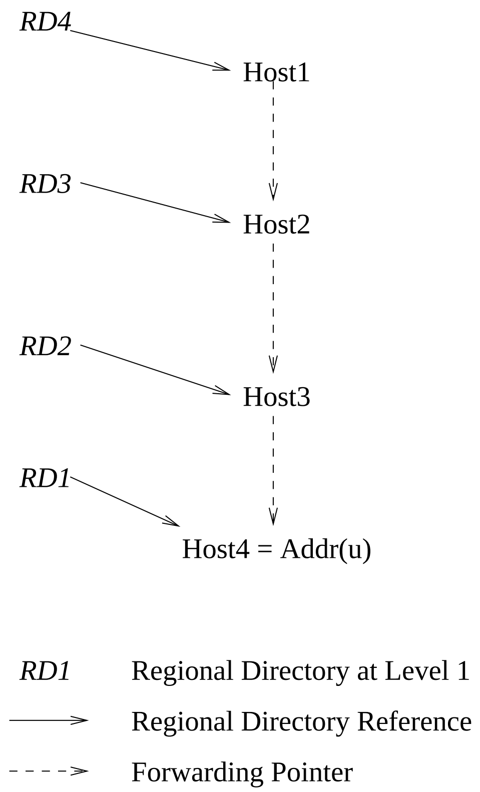


Figure 6: An example of a distributed Regional Directory.

system to hosts within the same region. This ensures that migrations to nearby locations cost a smaller amount that large distance migrations.

A regional directory is a hierarchically structured distributed directory that utilises forwarding pointers. Each directory must have an entry for each vertex or host within the regional directory or a mechanism for locating the host. For a hierarchical regional directory structure, there exists a regional directory for every level within the hierarchy which must maintain references for each host.

In a regional directory system, some regional directories will have direct pointers to an object, while others (of a higher level) will have indirect pointers that form a chain pointing to the object. Figure 6 shows an example hierarchy which exploits regional directories and indirect referencing. Regional directories at levels 4, 3 and 2 contain incorrect references to the host corresponding to the host identified by u. The incorrect references are translated to forwarding pointers while the regional directory at level 1 contains a direct reference to the correct host.

Awerbuch and Peleg [6] have shown that the size of a hierarchical directory structure is bound by the number of hosts and the number of hierarchies involved. As the hosts within the system are limited this cost has an upper bound. Awerbuch and Peleg have also shown that the communication overhead of the locating mechanism is within a polylogarithmic factor of the lower bound.

Regional directories supports a version of forwarding pointers to support transparent relocation. If an object migrates outside of its current region then full updating of the directory structure is performed; if the migration is within the same region then forwarding pointers are used within the region. In this manner, resolution through nodes outside of the mobile object's region can then follow the chain of forwarding pointers within the region to obtain a reference. This chain is always limited to a single region. The potential for cyclic forwarding pointer chains is limited by the introduction of checks which cut the chain at points of duplication.

2.4 Distributed Object Systems

A distributed object system is an object system where the objects are distributed over multiple virtual, or actual, processing nodes. Communication between the distributed objects is often performed using forms of remote procedure call or message-passing systems.

Naming systems are used in distributed object systems to support name binding at the object level of the system; the level of objects (fine-grained or course-grained) and the level of naming required can vary with each system. The level of object granularity may depend on the type of distributed object system itself; for example, a client/server system deals with course-grained objects, while a distributed shared object system often deals with fine-grained objects. It is common, within distributed object systems, to utilise a naming system as the basis for resource discovery mechanisms such as trading, registry and metacomputing services.

Distributed object systems may include support for mobility. However a distinction is made between object systems that are primarily distributed with some support for mobility and object systems that are primarily mobile, although some of these systems may support distribution paradigms.

Distributed object systems are relevant to this classification as they often provide sophisticated naming systems and have varying support for location transparency, location independence and relocation transparency. Accordingly, these systems also exhibit varying support for migration. Location transparency is important especially for resource discovery in middleware systems, where the scope of distribution is wide and the location of a service is unknown or changeable. The ability to refer to an object through a location independent means allows services to be found regardless of their current location or state.

The naming systems for the distributed object systems presented in this section are examined with respect to their ability to manage relocation and migration of objects in addition to pertinent issues such as their scalability and name system usage. Specific attention is given to ORB systems that provide support for naming systems and/or mobility, and to naming systems that are used to provide resource discovery mechanisms.

2.4.1 Aleph

The Aleph Distributed Object System [43,84–86] supports mutually exclusive referencing for distributed and mobile shared objects within a distributed system. The distributed system consists of multiple Processing Elements (PEs), which are Java Virtual Machines [180]. Each PE within a distribution group knows about each other PE in the group, with information replicated and updated within the group upon PE initialisation. Objects are relocated in response to explicit client requests for exclusive access.

Names that reference distributed shared objects are bound to the channels connecting those objects. A name consists of a tuple of the source and destination objects and exists only for the time that the communication exists. Names are globally unique within the distributed object system.

Aleph provides two mechanisms for locating objects. The first is a home directory scheme where a home PE is designated for each object and is updated upon relocation. This home directory is contacted by other objects to obtain the current PE of the mobile object. The second mechanism is the Arrow directory protocol [43, 85] which is unique to the Aleph system. The Arrow distributed directory protocol is a scalable and localised mechanism for ensuring mutually exclusive access to mobile objects. The Arrow directory service has two main facilities: it provides the ability to locate a mobile object (resource discovery) and it ensures mutual exclusion in the presence of concurrent client requests. This provides a synchronisation mechanism which is required as client requests within the Aleph system are to instigate mobility. This requirement is not so useful within a ORB system unless a service explicitly requests that it will serve requests on a single client basis.

Each PE contains a directional pointer that corresponds to the path required for access to an object. Each name, when used as an index, provides a directional indicator which can be used to locate the object. To achieve exclusive access, as each index is accessed, its directional pointer is reversed so that it now points in the direction of the requesting object which will subsequently possess the mobile object upon path completion. Figure 7 shows an example of the Arrow directory protocol in action. Initially a mobile data object is held at the PE identified by B (i). When a *find?* request is sent by PE A, the directional pointers at each PE are flicked to point at the source of the request, (ii) and (iii), so that when the mobile data object is found and relocated, the directional pointers are already correctly positioned. If another PE, for example C, concurrently requests access to the mobile object it is diverted prematurely to PE A where access blocks until the mobile data object arrives.

The Aleph system exploits a combination of the Arrow distributed directory protocol and the home directory protocol. This combination of mechanisms uses the home directory

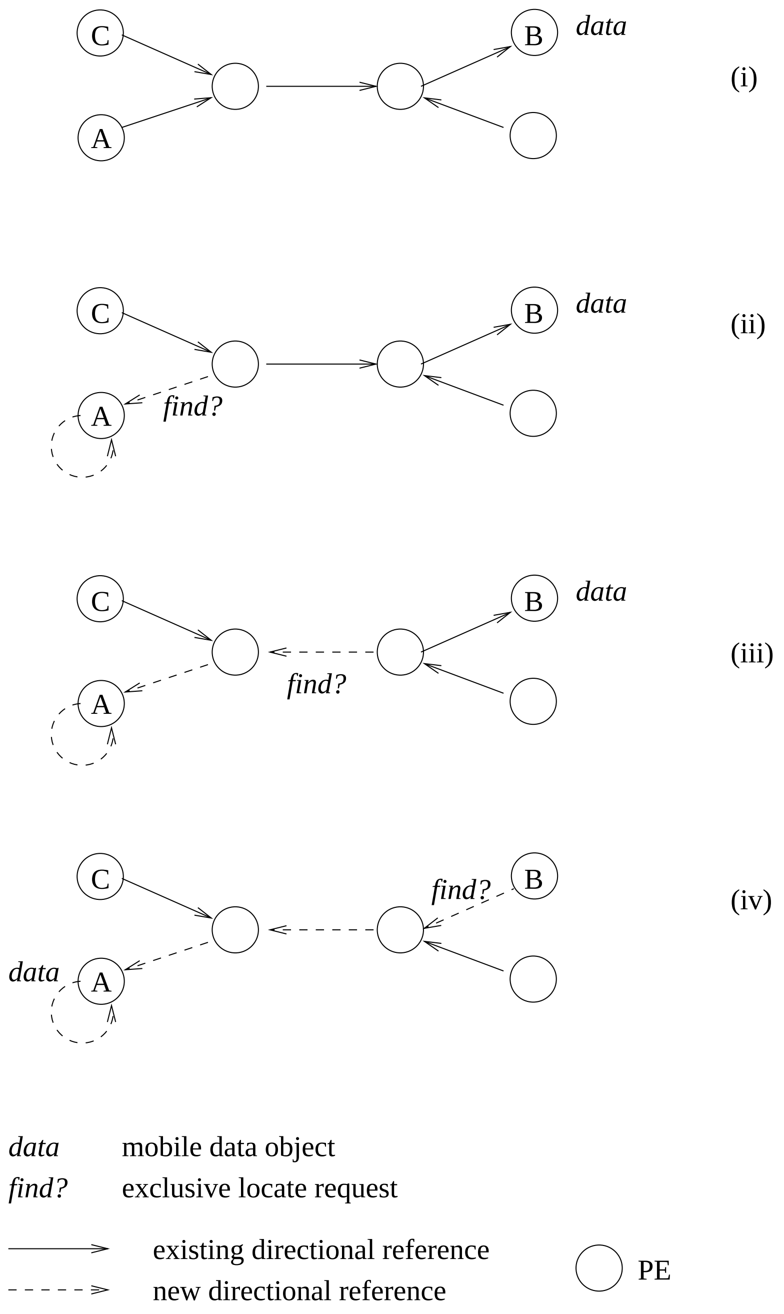

C
B
data
(i)
A
C
B
data
(ii)
A
find?
C
B
data
(iii)
A
find?
C
find?
B
(iv)
data
A
data mobile data object
find? exclusive locate request
existing directional reference
new directional reference
PE

Figure 7: The Arrow distributed directory protocol used within Aleph.

as a backup mechanism with the additional concept of best-case paths between a client's home location reference and the current location of the mobile object. Demmer *et al* [43] describe the potential distribution of the directory structure through a two-level structure where small network neighbourhoods have local home location directories and the Arrow directory structure operates at directory level granularity.

2.4.2 CORBA

The Common Object Request Broker Architecture [18,136,138] (CORBA) is a specification for an ORB architecture which provides language independent support for distributed client and server objects. CORBA has been developed by the Object Management Group (OMG) with contributions from many researchers and developers within the distributed computing field. It has roots in work produced by the Open Group [140], DCE [109] and Ansaware systems [17, 28].

The ORB defined in CORBA acts as a simple registry and naming system. The CORBA specification outlines many additional services [136] which may be accessed through the CORBA ORB architecture, one of these is a naming service [135] which provides contextual name location and management. CORBA has been designed to be extensible so that anyone can build a service and integrate it as part of a CORBA implementation.

The CORBA specification defines an Interface Definition Language (IDL) [136], which is used to specify the interfaces of service objects in a language neutral manner. An object implementation can then have stub references created in any target language that the client requires by using this IDL specification. CORBA IDL can be used as a wrapper over existing code objects. The ability to wrap external objects provides support for legacy objects, which may not necessarily be object-oriented in design, to be integrated into a distributed object system. Each server object registered with a CORBA system is given an object adapter; these object adapters act in a similar way to a scion or skeleton.

CORBA provides an implementation repository which is used to store interface information for objects registered within the CORBA system. Each object type has a unique (for that repository) identifier which enables clients to look up details of the interface in a method similar to reflection [105, 106]. CORBA objects also have unique names which are termed Interoperable Object References (IORs). Each IOR consists of information about the remote object, its adapter and specific protocol details. The implementation repository can be used to map a type description to an IOR.

CORBA supports a trading service [137] that allows clients to select an object reference using a semantic description. This trading service can be used to extend a CORBA system from a single level directory to a two-level directory structure. The idea of an integrated trader system or federated traders has also been proposed, allowing the possibility of multi-level CORBA systems.

CORBA does not specify a peer-based or hierarchical structure of ORBs. CORBA ORBs can intercommunicate outside location domains but often using implementation specific mechanisms, such as broadcasting within subnets or defined network subgroups to locate other ORBs. Requests for server references are then broadcast to all other known ORBs within the system and ORBs intermittently *ping* each other to check for liveness. Trading services can be used to link ORB domains together.

Object location in CORBA is performed using one of three methods: using the implementation repository, access through a specified IOR, or through a trader mechanism. These mechanisms provide varying levels of flexibility and ambiguity, with the least flexible and most efficient being direct access through an IOR. With a trader system, simple attribute ordering is available organised by the requesting client through the use of a constraint language [137]. Furthermore, the client may set certain attributes in its request to indicate how name resolution should proceed. These attributes include how many external traders should be contacted, how many responses should be returned, and how many preferences it would like matched.

The MASIF (Mobile Agent System Interoperability Facility) specification [128] is a CORBA extension specification provided by the OMG which describes a standard technique for developing mobile object integration services under CORBA. It is based on experiences with several existing mobile object environments, including Telescript [48, 120], Aglets [114,115] and d'Agents [72]. The MASIF specification outlines common concepts such as object, host, place and location in a standard way which can be mapped onto existing mobile object systems. The MASIF specification does not outline how client references and registry information are to be maintained. However, it has been designed to support common relocation schemes such as the home location model, forwarding pointer model and a central registry, which may be used by the mobile object system.

Schmidt [160] proposes a method for relocation based on an existing CORBA mechanism called a *location_forward* message. These messages are sent to a client if the server object which was matched by a registry did not match the object defined in the request. Schmidt proposes using these to update locations in connected clients which requires a forwarding pointer as a residual object to send *location_forward* messages. No method has been proposed to deal with the case of unconnected clients. Currently, CORBA can not support mobility between location domains [83].

2.4.3 DCE

Distributed Computing Environment [108, 109] (DCE) is a simple middleware layer that supports distributed computations in a client/server model. DCE was designed under the aegis of the Open Software Foundation (OSF) and has been designed as an extensible framework for client/server communication supporting the dynamic addition of services.

DCE services include a security service, cell directory services, global directory services and distributed time and file services. A DCE cell is used as a registry service which maps names to objects. Within a cell, there may exist several cell directories which are replicas of one master cell directory. The master cell is the only directory that allows changes to its data; all changes are then broadcast to the replicas. A DCE global directory service provides mechanisms for names to be resolved outside of the local cell.

Names with DCE are *binding handles*, which contain location dependent information. When a client wishes to locate a server reference in DCE it must either have a binding handle through which it can directly reference the server (static binding) or it must contact a directory service (dynamic binding). The directory service may be either a cell or a global directory. A hierarchy of global directory services and cell directories represent a distributed and replicated directory service that caches object location data.

The global directory service provides a resolution service for cell names, allowing a client to access cells outside of its local cell. The global directory does not perform any object name resolution but instead hands out references to cell directories that are able to perform object name resolution.

When contacting a directory service, a client must use an interface name to select a server of a specific type or an object identifier to select a particular server. If contacting a directory service, the client must wait until the directory service returns a binding handle and is then able to use the returned binding handle to connect to the server. Binding handles can be stored in a string form which can then be reused for static binding by storing them directly in the client code.

A trader has been developed for DCE systems as a service that enables clients to search for objects matching a semantic description, and to perform name resolution between DCE cell directories. The DCE trader is based on the Open Distributed Processing (ODP) trading function developed as part of the Basic Reference Model of Open Distributed Processing (RM-ODP) [101].

Location using a DCE trader service is performed by matching type specifications provided by clients against type specifications provided by each registered service. After obtaining a set of exact matches, a trader then applies additional preferences that the client may have specified. The client will receive a set of interface names that indicate the matching services found by the trader. The client then has to contact the naming service to obtain the host and endpoint information before contacting the object and binding to the name.

DCE does not provide any support for mobile services. It does provide support for relocation in case a server moves through direct user intervention before or during an existing client connection. If a client loses its connection to a server object, it must contact the relocation service to obtain a new binding handle, however any pending requests are lost unless explicitly stored by the server object.

2.4.4 DCOM

The Distributed Component Object Model [27,155] (DCOM) is a middleware system based on DCE produced by Microsoft Corporation. DCOM is an extension to the COM [127] system, designed to support non-distributed object services and is similar in concept to CORBA and Java RMI.

Location transparency is provided through the use of *monikers* as names. Monikers are objects that manage all referencing and communication detail on behalf of clients. Monikers support a common interface and have a client-readable form (a URL) which does not preserve location transparency. Unique identification is performed through the use of DCE-based GUIDs. Although similar in concept to RPC-style and SSP chain stubs, monikers are more complex and provide more functionality to the client.

Monikers contain all information required to locate their referenced object, either through an active connection or by contacting a local name server. Monikers also contain information that can be used to create an object (retrieving the object from the DCOM Class Store) if required. Remote communication is performed using DCE-based RPC; stub and skeleton code is produced using an IDL (MIDL).

DCOM provides a distributed name server through distributed Running Object Tables (ROTs). A ROT maintains a list of currently registered objects and information on other, remote ROTs. This model is similar to that used in d'Agents and MOA. The structure of the distributed name server system is configured through client interaction.

DCOM libraries probide facilities to create objects on remote nodes. This requires the specification of locations, and can not be performed through colocations. DCOM does not provide explicit support for object migration, but services can be migrated in response to node failure. Moniker connections can also be redirected to other existing servers in response to server failure. DCOM does not support relocation transparency. Because of the complexity and functionality already provided by monikers, it is expected that these mechanisms could be extended to support relocation transparency and object migration if required.

2.4.5 Globe

Globe [8,10,176–179] is a distributed metasystem designed to support a global name space, object replication and frequent object migration. Of all the systems examined in this chapter, Globe is the most similar in ambition and concept to DISCWorld. Globe uses a distributed shared object model, where a local object acts as a proxy for communications in a similar way to a fragmented object model.

Globe uses a two-level naming system that initially maps a readable pathname to a set of object handles that represent matching objects. The second stage of the naming system resolves these object handles to context addresses, which can be used to directly communicate with an object. A context address specifies where to locate an object (the IP address and port number) and also the protocol through which to contact the object. An example of a protocol specification is a Uniform Resource Locator (URL) [19].

The two-level naming system has been designed to minimise the impact of mobility and location updates throughout the system. Many mobile object systems with distributed naming services suffer from coherency problems when updating distributed databases, or use mechanisms such as forwarding pointers or home locations as has been previously mentioned. One of the main reasons for the use of these mechanisms is that it is difficult to keep track of every reference that has been provided and to update each reference upon migration. A two-level system overcomes some of these difficulties as object handles, which are location independent, are used as intermediate references until the reference is resolved to a contact address. This concept is similar to that of state-based references as used in the DISCWorld ORB system and handles as used in the fragmented object model. In these models a location independent object is used throughout part of a communication allowing coherency issues to be limited to specific databases.

Globe supports its two-level naming system by using two services: a naming service and a location service. The naming service is responsible for mapping symbolic names to object handles and the location service is responsible for mapping object handles to sets of contact addresses. Hence, it is only the location service that needs to be maintained during migration.

Globe utilises a wide-area location service to manage name resolution and object migration [9]; this location service utilises the principle of locality to optimise resolution and access. Resolution occurs using a distributed structure based on geographical domains in a tree-based hierarchy. Contact addresses are stored in leaf nodes within the tree structure; forwarding pointers to subnodes are stored in parent nodes. For example, Figure 8 shows a tree structure for the storage of the contact address of object *A*. Parent nodes of the leaf node *N4* contain forwarding pointers to the correct subnode which contains *A*. Hence resolution of an object is directed from the root node of the location service to the correct leaf node. Each parent node above the leaf node will contain a forwarding pointer that will direct any resolution to the correct subtree. Any migration of objects will result in a tree traversal, reversing the forwarding pointers to the correct direction or removing in a similar fashion to Aleph's Arrow protocol.

To increase potential optimisations through locality, contact addresses for objects which are geographically close are placed within the same subset of the root node, *i.e.* the same distributed root node. To correctly place each object in the same subset, a location mapping table is maintained which maps a location into the appropriate square of designated area.

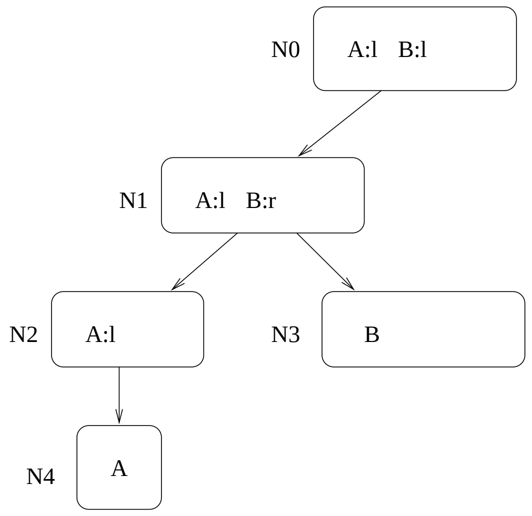


Figure 8: The tree storage mechanism for Globe's location service.

2.4.6 Globus/Nexus

Globus [42,54,57,58] is a global distributed object environment which uses a metacomputing directory service based on the Lightweight Directory Access Protocol [186] (LDAP) as its registry and naming system [54, 182]. Globus uses Nexus [59] as its underlying communications tool. Nexus is a distributed message-passing library that is based on Active Messages [181].

Globus has been designed to provide information rich dynamic configuration of resources, taking advantage of the dynamic nature of networks and network loading. The core of the Globus project is the Metacomputing Directory Service (MDS) [54]. Based on LDAP, the MDS provides a uniform and well known interface to information regarding registered services.

Names are organised within the MDS as a directory information tree. Objects are named using a globally unique name termed a distinguished name. A distinguished name consists of the path from the root of the directory tree to the specified entry. The directory information tree is constructed in a similar way to URL addressing, with objects organised into hierarchies based on their country, organisation and further organisational units. Distinguished names are not location transparent as one mandatory attribute is the host and entry point information for the object.

Location of objects within Globus is performed using the LDAP mechanisms for attribute matching and searching [182]. Attributes associated with an object can be organised into mandatory and optional groupings, additionally certain attributes are guaranteed to exist within the directory information tree. Entries in the MDS contain a time-to-live attribute which defines the time limits within which the data can be cached. This facility is an extension to the standard LDAP implementation.

The Globus implementation of MDS extends that of LDAP by allowing contextual limitations to information progression and providing client-side caching which removes the need for a LDAP call for each local access. These extensions make local accesses more efficient than in the standard LDAP implementation [54] while still exploiting the global nature of LDAP.

Globus/Nexus does not directly support mobile objects and hence does not support relocation. By allowing cached, and potentially out-of-date, information to exist within the directory structure, the MDS provides a form of forwarding pointer. This facility is not suitable for mobility support as it incurs high access costs for systems with frequent object migration.

There exists several implementations of mobility based on Nexus [60,132]. NeXeme [132, 133] is a Scheme [153] based dialect which supports native mobility primitives. In NeXeme, a mobility counter is used to maintain reference counting (NeXeme is primarily designed to support distributed garbage collection [147]) and forwarding pointers are used for reference relocation. Forwarding pointers are lazily evaluated upon direct access to reduce the length of the forwarding chain.

2.4.7 Hobbes

Hobbes [121, 163] is a distributed object system and an extension to the SSP chain mechanism (as described in Section 1.1). Hobbes is designed to support flexibility in client/server communications. SSP chains define a relocation mechanism similar to a forwarding object model with additional optimisations. A stub is used as a client side reference, each stub has an matching scion on the server side. When a server object migrates, it creates a new SSP, linking a new element into the chain.

Hobbes utilises several forms of references above those defined in the SSP chain model including bindable and continuation references. Bindable references are similar to unconnected references as described in the transparent relocation model proposed in this thesis, and can only be used to connect or unconnect to a server object. The reference contacts a binding object to perform connection operations. A binding object also exists on the server side and is responsible for deciding whether the connection is to be permitted. In Hobbes, an unsuccessful connection attempt can be forwarded to another binding object by the rejecting binding object.

A successful binding will return a continuation reference to the client. A continuation reference can be used to invoke methods on the server object in a manner similar to RPC. Each server object has a remote interface defined in an IDL.

The use of smart binding objects allows Hobbes to be used as a framework for adaptiveness and increased server flexibility. For example, a binding object can be programmed to provide references for existing or previously known clients and to pass on binding requests to server objects of a newer version for new clients.

Hobbes extends the SSP chain mechanism through the addition of a name server. Server objects can register their interface with a name server which can be queried by clients for an interface name match to perform object location. The name server is a centralised server and is designed to support small scale distributed object systems. Relocation is performed using the SSP chain mechanism.

2.4.8 Infospheres

Infospheres [33,34] is a distributed object system designed to support worldwide networks of resources. Resources are defined as objects, machines or people. Infospheres defines *personal networks* which can be used to define communication pathways and process organisation patterns. A personal network consists of a set of processes and communication channels that connect these processes. Communication is performed through a message-passing model with channels connecting inboxes to outboxes. An object may have multiple inboxes and outboxes through which it can access other objects.

Objects within Infospheres are any resource that can be used as a process; *i.e.* any resource that can support the infosphere model of message-passing communication through inboxes and outboxes. Each object has a home location that maintains an inbox that is responsible for handling reference requests. It is this home location that is initially obtained as a reference.

A name bound to a process object consists of a description of the inheritance relationship of the object and its interfaces. This description can be used to match service types and to verify that two objects are able to communicate. Assuming that the required inheritance relationship for an object can be established, these relationships can be encoded in URL-form and resolved through web access [34]. The assumptions here are that the URL can be constructed in the first place and that the home location is always available. Mani Chandy *et al* [34] are investigating distributed directory services to enable metacomputing-based services within Infospheres. The usage of a home location for each object introduces limitations discussed in Chapter 1 such as the trombone effect.

Mobility can be supported by using the home location as an update point. During migration when a process is *frozen*, all inbox references become out-of-date; after migration is completed inboxes are again able to accept communications. Due to this restriction relocation is not a requirement as Infospheres objects are only ever in the unreferenced or connected states. By updating the home location, migration can occur between sessions.

2.4.9 Java Remote Method Invocation

Java [55, 167] is an object-oriented language that supports distribution and distributed communication. Remote objects and Remote Method Invocation (RMI) [167] allow Java objects to perform distributed computation and communication. Java provides additional distributed object services independent of RMI, such as a messaging service, the Java Naming and Directory Interface (JNDI) [168] and Jini [169–171].

The Java RMI system is a three-tier communications protocol. Clients communicate with servers using an intermediate registry which is able to locate objects and store bindings between names and server objects. Names within a Java RMI system are in the form of URLs which contain information about the host, entry point and object name. A name binding is requested by the server object upon registering with a local registry. Java RMI does not support a hierarchy or grouping of registry objects. A registry can act as the registry for multiple hosts, accepting registrations from server objects within the group providing limited scalability.

Server objects define a remote interface, which can be exported to remote clients to define allowable remote communications. A client must have access to, or dynamically download, a stub which will act as a reference to the remote server and provide access to the methods defined in the remote interface.

Location is performed by requesting resolution of a name from the remote registry that registered the server object. A simple table lookup on the name is performed and the matching stub is returned to the client. Java RMI provides mechanisms for constructing URLs to represent a name for a server object.

An extension to the Java core API proposes a representation for Java objects including Java RMI objects in an LDAP directory [157]. This proposal outlines an attribute-based naming schema to be used with LDAP directory structures for Java serialised objects, Java RMI objects and JNDI references [168].

A Java RMI system that utilises LDAP for its naming may use this mechanism to provide a scalable service as it can register its objects within a global, scalable directory structure. This solution is yet to be tested fully; in particular the impact on large amounts of out-of-date data if objects are frequently replaced (for example, by an object of a newer version) remains unknown.

Weak mobility is a natural ability of Java objects as they can be easily serialised and transmitted between Java virtual machines, however mobility is not something which is provided in the standard Java programming environment. Strong mobility is restricted by the limited ability to access both data and execution state, particularly when Java objects are interpreted. Java does not provide any support for relocation. Many mobile object systems have been developed using Java's facilities including [1, 14, 48, 72, 87, 95, 115, 185].

Relocation services have had to be developed to support each system as discussed in Section 2.2.

2.4.10 Legion

Legion [73–75] is a large scale distributed object system designed to support a large degree of parallelism. Legion has been designed to abstract over the details of physical locality to provide instead the view of a single, nationwide metasystem. To provide this abstraction, Legion provides support for managing scheduling of server tasks, data transfer, communication and synchronisation.

Objects within Legion are service objects that are contained within an independently defined wrapper object. The ability to wrap objects and provide a consistent interface enables Legion to easily integrate legacy objects. In the rest of this discussion, a Legion object should be interpreted as the external, wrapping object rather than the contained service object.

Legion objects contain four main elements: an address space, a name, a class and a set of capabilities. Each object has an interface defined in IDL (either CORBA IDL or MPL [76]); these interfaces enable external clients to create queries and method calls regardless of the contained object's implementation language.

Legion has a two-level naming system, similar to that found in Globe and Globus. Each object may be given a symbolic context name which is easily understood by clients. Through access to a name server, this symbolic name can be translated to an object ID which can then by resolved to a reference. Symbolic names are location independent and transparent, while ID's are not. Symbolic names can be used to provide a location independent, and more easily interpreted, abstraction for an ID.

Legion uses a distributed location scheme with multiple distributed name servers. Each name server uses a caching mechanism and access to defined resource objects for each server to provide name resolution. Each server object has resource objects consisting of its *binding agent* and *metaclass*.

For example, a client wishing to resolve a name to object A will contact its local name server and request resolution. If the name server has a cached reference it will return that, otherwise it will contact the binding agent for A. This binding agent is a class associated with A and may have access to the location of A. If resolution is still not achieved, then the binding agent requests resolution for the metaclass of A, the metaclass will be able to resolve down to A eventually. This procedure goes on, requesting resolution from metaclasses further up the class tree until a request attempts resolution on the *Legion* class. The Legion class is the root (local root) of the class system and knows how to resolve each object. This class has a predefined and guaranteed location.

This context-based searching takes advantage of the principle of locality of reference. For example, given an object A, with a context name *math.symbol.tex.pi* that is attempting to resolve a name *math.symbol.latex.pi*, A will contact its name server, and if resolution fails, will proceed through the context objects associated with itself. Once it proceeds to the context object associated with *math.symbol* it will then be able to resolve down the hierarchy to obtain a reference to the required object. In this way, objects that have been created by the same user and exist within the same context will have an optimised resolution path.

2.5 Summary

The mobile and distributed object systems examined in this chapter have differing models for naming and reference management. In some systems transparency and independence are of utmost importance, while in others they are considered unnecessary. Some systems recognise the importance of transparency but then sacrifice it for efficiency of object location. This can then limit the scale or opportunity of migration that is allowed within the system.

Tables 1 and 2 summarise the support for reference independence and transparency present in the mobile and distributed systems examined in this chapter. Each system is classified according to its support for location transparency and independence, relocation transparency and migration transparency.

The location and relocation mechanisms used within mobile and distributed object systems are similar. Existing methods (such as the home location, forwarding location and centralised registry models) are used frequently and with varying degrees of success dependent on the nature and use of the system. Often a combination of methods is used, such as using a forwarding location as the default and then using a broadcast or home location model as a backup. The combination of models requires multiple mechanisms to be supported and the maintenance of information that may never be used.

Table 3 summarises the relocation mechanisms found within those systems that support relocation and indicates whether the mechanism is transparent. As can be seen in Table 3, many systems provide a means for relocation transparency. However, the mechanisms used are often based on location dependent referencing or the use of residual objects. The forwarding location model is a common model, additionally many optimisations of the forwarding location model have been introduced.

Distributed registry systems have been developed in several distributed object and mobile host systems. These systems provide a means for locating distributed objects using a symbolic location independent name, but these systems do not provide relocation transparency or mobility support. Distributed registry systems provide mechanisms for location independent resource discovery; these mechanisms are often missing in mobile object systems. The ability to provide an object location based on a semantic description of a server object is also missing from many mobile object systems.

To produce a complete model for location and relocation transparency, the correct naming model must be decided upon. To be considered are the choices between using location dependent or independent names, the provision of location transparency, and the effects these choices have on relocation transparency.

System	Location Transparent	Location Independent	Relocation Transparent	Migration Transparent
Ajents	$\times$	$\times$	$\times$	$\times$
d'Agents	$\times$	$\times$	$\times$	$\times$
Emerald	$\surd$	$\surd$	$\surd$	$\surd$
MOA	$\times$	$\times$	$\surd$	$\surd$
Obliq	$\surd$	$\surd$	$\surd$	$\times$
Sumatra	$\surd$	$\surd$	$\times$	$\times$
Mobile Process	$\surd$	$\times$	$\surd$	$\surd$
Regional Directories	$\times$	$\times$	$\surd$	$\times$

Table 1: Transparency and independence within mobile systems.

System	Location Transparent	Location Independent	Relocation Transparent	Migration Transparent
Aleph	$\surd$	$\times$	$\surd$	$\surd$
CORBA	$\surd$	$\times$	$\times$	$\times$
DCE	$\surd$	$\times$	$\times$	$\times$
DCOM	$\surd$	$\times$	$\times$	$\times$
Globe	$\surd$	$\times$	$\surd$	$\times$
Globus-Nexus	$\times$	$\times$	$\times$	$\times$
Hobbes	$\surd$	$\times$	$\surd$	$\times$
Infospheres	$\times$	$\times$	$\times$	$\times$
Java RMI	$\times$	$\times$	$\times$	$\times$
Legion	$\surd$	$\times$	$\surd$	$\times$

Table 2: Transparency and independence within distributed object systems.

System	Relocation Mechanism	Transparent?
Ajents	Forwarding reference.	$\surd$
Charlotte	Mixture of forwarding pointer and an update message sent through duplex connections.	$\surd$
DEMOS/MP	Forwarding pointer chain with lazy evaluation.	$\surd$
Emerald	Forwarding process.	$\surd$
Globe	Geographically sorted naming tree.	$\surd$
Hobbes	SSP chains.	$\surd$
Infospheres	Home location.	$\times$
MOA	Multiple schemes through a caching mechanism that indicates the type of scheme to be used and any information to support the scheme.	$\surd$
Mobile IP	Home location and broadcast.	$\surd$
MOSIX	Home location process with fast link to mobile process.	$\surd$
Obliq	Forwarding objects and centralised name server as backup.	$\surd$
Regional Directories	Forwarding pointers.	$\surd$
Sprite	Home location.	$\surd$
Sumatra	Forwarding pointer that returns exception with new location of object.	$\times$
V-System	Transparent mapping of client references involved in a communication, and broadcast to update.	$\surd$

Table 3: Relocation schemes used within distributed and mobile object systems.

Chapter 3

Naming and Naming Models

Within a mobile or distributed object system, a client must be able to resolve the name of a server to reference the server. A server must be able to bind a name to itself through some mechanism. Often these facilities are provided as part of a language or system at a low level through mechanisms such as declaration of object variables or, in the case of CSP [88], specification of named tasks. When dealing with a distributed or mobile object system naming becomes a more difficult problem as naming has to occur between large scale objects that may be unrelated in terms of their language or support system. In these systems, naming occurs between objects that must learn about each other dynamically and perform communications. In order to overcome these difficulties, naming is often addressed by some sort of naming system.

Naming an object within a large scale object system prompts the questions: "How can I ensure that my name is unique?", and "How can I publish my name beyond my home system?". These issues have been tackled by many naming systems and models [15,26,135] developed specifically for distributed object systems and mobile systems.

There are many different views on how a naming system should operate and, indeed, what naming means. Goscinski [68] states "*that every entity in a system deserves a name*". Some systems take the view that every item down to the variable level is named, while others take the view that only large objects or, indeed, accessible processes deserve to be named. The level of naming involved in a system is related to the purpose of the naming system; for example, a naming system to support a distributed file system has names for every object (file) within the system - naming will not be at the level of high-level processes [158,174]. Accordingly, a naming system to support a large scale, widely distributed object system for client/server communication should provide naming services at a high level for server and client objects but not necessarily for internal variables or objects [135].

Names can also correspond to the communications channel linking two objects instead of the objects themselves [43,84–86]. For example, the Aleph Arrow directory protocol [43] uses names that correspond to a directional pointer to follow to locate an object. These names consist of a tuple: $(source, object, dest)$, which defines the direction of travel as well as the object to be found. Names in this category are globally unique within the system and can be used to support directional directories [56, 84].

Additionally, there is much argument about the scope of visibility of names within a distributed object system, *i.e.* local versus global names, and whether names need to be unique. Names can also be contextual. For example, a name may be a unique composition of contexts that decrease in scope until a single entity is described. This mechanism enables local names to be used by hierarchical description, *i.e.* a single name does not have to be unique within a global perspective but its location or semantic scoping describe its true global name [135, 150].

Local names are proposed by [56, 150] with a good naming scheme defined as one that exhibits the following properties:

Locality: local names are used to refer to local objects. Local ambiguity is avoided by the use of different names for every different object.

Mechanisms for Name Communication: each object understands what every other object is by name and understands aliasing. If an object receives two object references that reference the same external object, and each object reference has a different name, then it is able to equate the two names.

Location Independence: the naming scheme does not change (is independent) when objects are moved.

In such a system, a name would be resolved to an object by examining the context of the name and finding the local name that corresponds to the complete name (with context). In this way, each name must be unique within its local context and is made unique within a global context, and hence resolvable, by the addition of nested contexts.

Local names are managed within the system proposed by Flocchini *et al* [56] through naming by sense of direction. In this system, given an object graph depicting an active distributed object system, a name is an edge linking two objects. Figure 9 shows an example of this kind of system, where the domain of names possible in the system consists of the set of possible connections between objects. In this example, an object A has a reference to object B which is named (a, b). This name is unordered and is used as a reference by both sides of the communication. Hence, within this system, names are unique to each channel that links two objects as opposed to unique to each object or class of object.

Names can also be globally defined. A naming system with globally unique names must ensure that each binding is valid with regard to every name binding known to the system. A naming system with non-unique global names does not require validity checks for uniqueness but does have the problem of ensuring that all objects with the same name

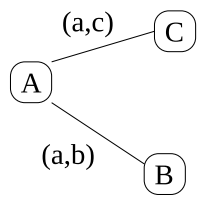


Figure 9: Direction-based naming.

support the same task or service. Management of names in this case may be determined by code reflection[3] or by some organisation responsible for equating and naming objects [57].

3.1 Name Structure

In general, a name is simply a string that is used to identify either an object or a set of objects. A name may have a defined syntactic structure which is dependent on the naming system. For example, a name may consist of several defining characteristics of its bound object combined to form a single string name. These characteristics may be the originating host name, the time of creation and a unique identifier for the object within the creating host's address space. A combination of this form is a globally unique name but is not location transparent.

A naming system that creates names according to well defined syntax rules is a structured naming system. Naming systems in this category give more information in their names and are used effectively to define a context mapping, however, they are also difficult to extend and use in different naming systems. Names in this form are often heavily syntax structured, with contexts being specified in a form similar to:

```
<context-name><separator><context-name><separator><object-name>
```

Figure 10 shows an example of a structured naming system that is represented in a tree structure. This example shows a possible hierarchy for a naming system for educational institutions based on a URL structure, where each node corresponds to an example syntax component. A particular institute is represented by a leaf node within the tree.

In the case where the tree mappings are changed, for example a new branch is inserted higher up in the tree, all names that are further down must be altered accordingly to show the new branching. For example, consider Figure 10 again, if a new tree node was created to split the *au.edu* branch into types of educational institutions, each name in the subtree has to reflect this change, as shown in Figure 11.

Context management has been a focus of research in the areas of language design and addressing schemes [63, 158]. Several techniques have been proposed to guarantee correct

[3]Code reflection enables a client to query another object for information on methods that the object provides or its inheritance hierarchy.

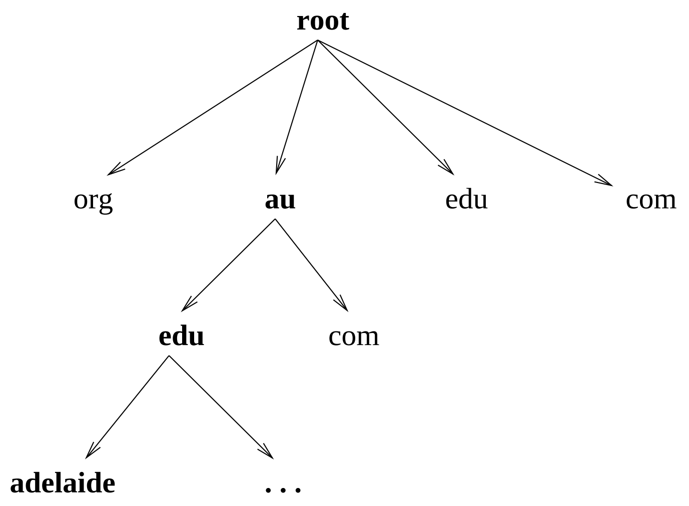


Figure 10: A structured, contextual naming scheme in tree form.

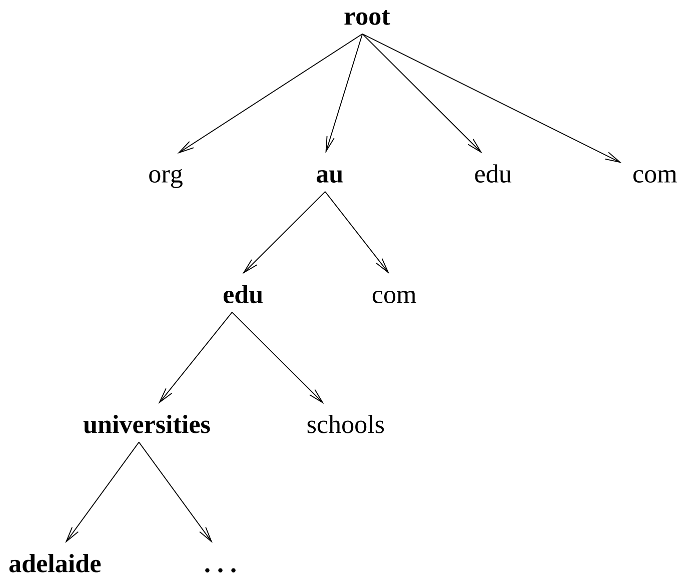


Figure 11: Dynamic extensions to a tree form naming scheme.

context resolution and to prevent naming conflict. Fraser [63] describes the common problem area of context management in language design and operating system design. Fraser proposes that context manipulation can be modelled abstractly by an extension to the lambda calculus [36] which models name/value substitution within named contexts.

Fabry [50] introduces the concept of using *capabilities* [44] as absolute addresses to objects within an addressing scheme. Instead of contextual names, capabilities that are generated by the system upon completion (or compilation) of code are used to access objects. Consequently, no contexts need be managed as absolute addressing takes place. Iliffe [92] proposes a scheme based on *codewords*, which are descriptive addresses maintained throughout the execution of a program.

Names that are not dependent on a defined syntax are termed structure-free names. Terry [174] proposes a naming system that supports structure-free names. This naming system allows the binding and resolution of names to take place independently of the syntax or structure of the names. By making the naming system modular and separating the tasks of the naming system from that of name construction it is possible to construct a flexible and extensible naming system.

A name can be viewed as an access key to a set of attributes, one of which is the reference or object to which the name is bound. Additionally, a name can be used to retrieve attributes provided by, and bound to, the object. A naming system can use these attributes to support intelligent name resolution as is found in complex resolution schemes such as trading schemes. The name binding system must provide a means to bind attributes to a name in a sensible and extensible manner for this to be accessible.

3.2 Naming Systems

Naming systems provide mechanisms for managing names within an object system. A naming system provides the support required for a server to obtain a name binding, to manage its binding, and for clients to resolve names to references. A naming system is also responsible for applying a suitable naming model to the bindings it is required to manage. Naming systems allow a greater level of abstraction in client/server communication by allowing clients to use indirect names to refer to a server or set of servers. The choice of naming system can affect the communication models available, with communication potentially being blocked until unbound names can be correctly bound to an object.

A naming system may be required to introduce new names into the system through server request, or it may be required to create new names appropriate to the naming model. A naming system may also be required to operate throughout a distributed namespace or with different naming models within the one system. It is because of these requirements that the development of a suitable naming system for a mobile or distributed object system can become a complex task.

Naming systems have been researched in the area of operating systems, involving process naming, as well as in object systems and mobility systems. Fundamental work in this area includes that of Fraser [63], Fabry [50] and Saltzer [158,159]. Naming systems are commonly found as the basis for registry and trading services within wide-scale mobile and distributed object systems.

A registry system provides a central, third party, service for server registration and server location, commonly found as part of an ORB system. A registry system typically makes use of a naming service to perform name management and validation. The design of the naming system has a significant impact on how the registry system works and what properties it exhibits [15, 16, 26].

Single level registry systems, such as those found in ORB systems [138, 167], provide a single client/server communication point and can often be the cause of communication bottlenecks in poorly designed object support environments. A single registry system provides limited robustness and failure management policies by acting as a single point of failure. Distributed or hierarchical registry systems [54] provide a more scalable mechanism for service registration and location but suffer from coherence problems. A distributed registry system may require that each registry contain all of the known references within the system, leading to a high communication load; or it may require that each registry know only about its own registered servers with unsatisfiable requests being passed through the hierarchy.

Trading services provide mechanisms to locate a name within a distributed system by using a semantic description of the server object [28, 54, 137]. Trading services typically provide mechanisms for creating a semantic description in an extensible manner and for resolving a description to the appropriate server object.

A trading service can be either a central repository or a database accessed by clients or by cooperating registry services. Integrated or federated traders can be combined to produce a distributed network of traders with an underlying distributed database [137]. A distributed system requires frequent updates to ensure that information is correct in each distributed database. Logically centralised systems, where the database is distributed and replicated over several nodes, suffer from coherency problems.

One of the main problems with trading services is determining how to specify that semantic properties of two objects are identical. Ideally, two servers that performed the same operation would be given the same name. When servers such as these are developed by different people and have different internal structures, it is difficult to automatically recognise the objects as equal. For example, it is difficult to specify which characteristics of an object need to be defined and which do not, and how these characteristics should be defined so that they may be compared. Additionally, a preference ordering within the specified characteristics must be defined.

The binding and resolution of names, either through simple matching mechanisms or through attribute-based searching mechanisms, are controlled by the naming model of the object system. The naming model of a system does not necessarily define the structure or creation process of names, but defines how those names are to be treated within the object system and what rules govern the binding and resolution of names.

3.3 Naming Models and Classification Schemes

Work on naming models and classification schemes has been divided into the areas of name binding and name resolution. Name binding is the area defining the rules by which a name binding is accepted into an object system. These rules may control such aspects as name uniqueness and mutability. Name resolution defines how a reference is obtained from a name. This process can be attribute-based, in which case it must be decided how these attributes are to be selected and ordered.

3.3.1 Name Binding Models

Name binding models must define the rules for managing and binding names to objects within an object system. When an object is introduced into the naming system it is bound to a name. This name can be defined by the object itself, being dependent on certain attributes of the objects such as its code-name, creation host and time of creation, or a name can be provided independently by the naming system. These bindings and the creation of names must be controlled by a name binding model.

Saltzer [158] defines several characteristics of a good naming system:

- Names must be location independent: *"... using physical locations as names guarantees that the context is limited"* [158]. With a limited context it is still possible to have name conflict, where references to multiple objects may refer to the same location, while only one object is resident at the location.

- Names must be able to be shared. A fundamental purpose of a name is that objects may refer to the same external object by the one name and hence accomplish efficiency in storage.

- Clients must be able to define private names for objects that are available only in the local context so that local names can be used in combination with global or universal names. This enables names to be resolved within a local context without the expense of global communication or context evaluation.

- Names can be from either a limited or an unlimited source. In a limited naming system, names are themselves in short supply and so must be bound carefully.

- A name may be a reference to another name, facilitating indirect addressing.

These attributes are specifically tuned to naming systems designed to support distributed file systems and addressing architectures. The naming system Saltzer describes in [158] is designed to support dynamic binding of code segments to support efficient code management and reuse within a low level addressing architecture and hence is also designed to support contextual definitions of variable names that may act as local bindings for objects. The issues raised in this kind of naming system are still pertinent to naming systems for mobile and distributed object systems.

Bayerdorffer [15, 16], based upon preceding work, defines six orthogonal characteristics that a naming system must define that extend those defined by Saltzer. This classification is initially defined for concurrent object systems. The characteristics are:

- Mutability of bindings: a name that may be rebound to another object is *dynamically bound* (**B**), otherwise it is *statically bound* (**b** or ¬**B**).

- Knowledge: a name system that may dynamically add names is said to have a *dynamic domain* (**D**), otherwise it has a *static domain* (**d** or ¬**D**).

- Multiply-bound names: if a name may be bound to multiple objects (*i.e.* sets of objects) it has the property of *multiplicity* (**M**), otherwise it has the property of *singularity* (**m** or ¬**M**).

- Multiply-named objects: if an object may have multiple names, the name system supports *aliased names* (**A**), otherwise the name system supports *unaliased names* (**a** or ¬**A**).

- Name sharing: if a name can be shared between clients, *i.e.* multiple communications can occur over the same name, the name system supports *shared names* (**S**), otherwise the name system supports *private names* (**s** or ¬**S**).

- Descriptive names: names that can carry information about the state of objects to which they are bound are termed *descriptive references* (**R**), otherwise they are termed *nondescriptive references* (**r** or ¬**R**).

Mutability of names means that a name can be changed to be bound to different objects. Mutability is required when a name must exist for the lifetime of the object system, but the bound service may change. In a system that supports mutability, an abstraction must be provided that hides the current object bound to the name. This level of abstraction allows client references to remain valid throughout the mutation process. This can be done by associating a form of handler with the object that may take the form of an additional object [27] or be as simple as a port number that acts as a common contact point for all referenced objects. Most systems allow mutability of names, *i.e.* dynamic binding of names to objects, whereas some explicitly forbid it in favour of static binding. CSP and Ada [139] require statically named processes, with Ada specifying names in terms of their task identifier and entry points.

The characteristic of knowledge specifies that a system's name domain be able to grow. Names may be dynamically added to the system by dynamic creation or through parameter passing from outside the system. A naming system that does not support knowledge requires a static definition of names that the service will support and is termed a closed system. A dynamic domain is one that enables addition or removal of names; a static domain is one where no alterations are permitted.

Multiply bound names provide the ability for clients to broadcast communications to multiple servers through the one name. This characteristic can be extended to support replication schemes as a form of fault tolerance. In Bayerdorffer's model, multiply bound names or *multiplicity* defines that all bound objects are accessed as one and hence broadcast or replication is native to this characteristic. This characteristic does not classify systems where one of a group of bound objects is selected through name reference.

Multiply named objects take part in multiple bindings within the naming system. This form of aliasing allows an object to receive communications through multiple names; these names can be used to allow selective computation depending on the name used. For example, in a system where a name corresponds to a object name combined with a procedure name, a bound object may choose to accept some communications (*i.e.* of a particular procedure matching) while rejecting others. In a service-based system, aliasing allows an object to be part of multiple service provision groups.

Name sharing allows a server to have more than one client, *i.e.* the name can be shared between multiple objects. This property is common among ORB systems where a server can satisfy multiple requests through multithreading. Name sharing can prohibit determinism as multiple communications can occur concurrently from multiple clients, which may be undesirable [15, 16].

Descriptive names have the additional element of a description that shows the current state of the object. A naming system that supports descriptive names must support addition and modification of the description.

Not all of these characteristics are necessarily present in any one naming system, and the characteristics selected affect the communication models that the naming system supports[4]. There exists naming systems that exhibit all of the above properties, however to implement a naming system efficiently it is often better to support a subset of the possible characteristics. A naming system that has few or none of these attributes is termed a *weakly expressive*

[4]The attribute of aliasing or multiple naming of objects can lead to polling unless communication is only accepted if it can be satisfied. This can lead to blocking communications which is likely to be undesirable in high performance distributed object systems [15].

naming system; a naming system that has many or all of these attributes is termed a *strongly expressive* naming system.

CSP is an example of a weakly expressive naming system that only supports aliasing. Aliasing is supported in CSP by the specification of nondeterministic entries which may be used as aliases with specified acceptance permissions.

Ada's naming system supports aliasing by acceptance based entries in a similar manner to CSP. The shared names attribute is supported by allowing multiple tasks to share the same name (hence, naming is transparent and not channel-based as in CSP). Ada supports name access through the ability to dynamically create tasks. Concurrent C [64] also exhibits descriptive naming by adding the ability to define entries relative to some expression which may be state based.

Linda [65] is an example of a strongly expressive naming system that exhibits all of the specified attributes. Linda expresses dynamic binding by allowing the same name to to be applied to multiple messages, or *tuples*, which may have come from different objects. Multiple naming is also expressed as messages which may be received by more than one target object.

A particular naming system is defined as an orthogonal point within the space defined by the above attribute pairs. A name system in **BDMASR** or $\top$ is a name system with all attributes while a name system in **bdmasr** or $\perp$ has no support for the defined attributes. For example, a naming system defined as **BDmASr** is a naming system that supports mutability of bindings, a dynamic domain, aliasing and shared names. The example naming system does not support multiply-bound names (it has singly-bound names) or descriptive naming[5].

Bayerdorffer uses this classification scheme in his thesis to classify several concurrent object systems, before proposing his own, Associative Broadcast [15], which is an example of a $\top$ naming system. Few systems represent $\top$ systems, with the Linda system being an example. Bayerdorffer's classification considered concurrent languages and systems including CSP, Linda, Ada 95 and Associative Broadcast, and does not represent systems of an ORB model for widely distributed systems. A classification of ORB models and existing mobile and distributed object systems is presented in Chapter 4.

Much of Bayerdorffer's classification deals with names bound to communication channels. For example, a name can be bound to a procedure entry within a remote procedure call as a combination of the identifier of the object that will perform the remote computation, and the procedure identifier. Alternatively a name may be bound to a communication linking a client ID, a procedure ID and a server ID. This is not the model of naming that is explored in this thesis. The model used in this thesis is per object naming with the potential for channel based aliases.

[5]Interestingly, this is the classification for the Emerald [102] mobile object system and the Ajents [95] mobile agent system, as explored in Chapter 4.

As Bayerdorffer's system predates a classification of ORB systems it also does not perfectly fit the ORB model. All of the systems within Bayerdorffer's classification belong to a two party model or a client/server model of some kind. It does not take into account systems with a fragmented object model, that may have many distributed parts, with the possibility of multiple naming models within the one system. This thesis presents extensions to the naming model suitable for multiple party systems such as the ORB model in Chapter 5.

3.3.2 Name Resolution Models

Name resolution is the process of locating an object reference using a name or some description of the object. Resolving a name to an object, or to an object reference, can be easily performed in the case of correct and complete client and name system knowledge. If all objects can be represented uniquely by a name and that name is known correctly in all cases by a client, a resolution scheme consists of nothing more than a simple lookup table, matching the name against the object. This is the scheme taken for simple resolution schemes as used in addressing architectures [158]. Even when a resolution scheme is complicated by structured names that have to be resolved in a specific context, it is still straight forward to evolve a hierarchy of naming tables that match the appropriate context [158, 174].

Naming for distributed and mobile object systems is more complicated than these schemes as a single name may not be enough to uniquely specify an object. Additionally, a client may not have all of the information it needs, or may even have incorrect information. This situation can be improved by allowing the specification of attributes that assist in defining characteristics of an object; these are categorised as *descriptive references* in Bayerdorffer's scheme. For example, given an object whose purpose is to calculate a surface map from an array of point values, it could be annotated with the attributes *fast*, *Fortran* and *CM5* to indicate that it was a high performance solution, written in Fortran [40, 126] that can only be run on a Connection Machine CM5 [175].

This form of resolution is more complex than a name matching scheme, as the resolution system has to determine which attributes should be matched, which should be taken as mandatory, and the ordering of attributes with regard to preferences specified by the client. Furthermore, binding of attributes to names also has to be handled as these too may have preference orders or inaccuracies.

Bowman *et al* [26] define a name resolution classification scheme based on attribute specification and preference ordering. In this classification scheme, resolution is performed through the specification of attributes that define the required object. Bowman *et al* define several preferences that can be used to order these attributes with regard to their importance and value. A resolution system consists of a set of preferences and their order within the resolution system. The preferences listed are not the only preferences available but those commonly found within name resolution schemes. Bowman *et al* use this model to develop an attribute-based name server called Univers [25].

A preference is defined as an ordering of approximation functions, where an approximation function is a function that is used to select attributes according to some specification. The preferences defined by Bowman *et al* are divided into client preferences and database preferences. Client preferences are those preferences that can be expressed by a client, defining how they wish the results of resolution to be ordered. Database preferences are applied to the database, maintained by the naming system, that contains all of the names, objects and bindings known to the object system. Database preferences define preferences on how the database obtains or sorts its information. The client preferences defined by Bowman *et al* include:

Registered preference. The registered preference, denoted $\prec_R$, defines a preference for attributes that are guaranteed to be registered in the registry database over those that are not. The registered preference defines two approximation functions *open* and *closed* where *open* is an approximation function for the attribute of unguaranteed inclusion, and *closed* is an approximation function for the attribute of guaranteed inclusion.

A *closed* approximation function is a function that is used to select all attributes that are guaranteed. An *open* approximation function is a function that is used to select all attributes that are not guaranteed. A preference ordering is indicated by $\prec$, where the right-most approximation function is the preferred function. A preference for guaranteed inclusion over unguaranteed inclusion can be expressed as:

$$\mathit{open} \prec_R \mathit{closed}$$

The preferences defined by Bowman *et al* define a conceptual preference. For example, the registered preference is not necessarily always a preference for guaranteed attributes but is a preference defined on the ability to guarantee. A naming system that has both guaranteed and unguaranteed information is classified by a registered preference of the form:

$$\prec_R\text{: } \mathit{open} \prec_R \mathit{closed}$$

where the registered preference, denoted by $\prec_R$, is defined with a preference for guaranteed information, a *closed* approximation function, over unguaranteed information, an *open* approximation function. A naming system with only unguaranteed information is classified by a restricted registered preference of the form:

$$\prec_R\text{: } \mathit{open}$$

Mutability preference. The mutability preference $\prec_U$ defines a preference for attributes that will always describe an object over those that may change. An approximation function *static* represents static definition, while an approximation function *dynamic* represents dynamic change. This preference can be expressed formally as:

$$dynamic \prec_U static$$

Precision preference. The precision preference $\prec_P$ defines a preference for attributes that are unambiguous, *i.e.* those attributes that are defined for at most one object. Two approximation functions are defined: *unambiguous* and *ambiguous*, where

$$ambiguous \prec_P unambiguous$$

Yellow-Pages preference. The yellow-pages preference $\prec_Y$ defines attributes into classes that are either *optional* or *mandatory.* This preference places more importance on attributes that are mandatory and hence can be tested for each object, over those that are optional. This can be expressed formally as:

$$optional \prec_Y mandatory$$

Explicit preference. The explicit preference defines a client defined preference, and hence the approximation function is defined explicitly by the client. This allows a client to request a preference for an object with attribute A over attribute B, where the order can not be specified by any of the previously defined approximation functions.

These client preferences can be used to specify the importance of attributes in a client's requests, and are used to order the results of a resolution with respect to these preferences.

The database preferences defined by Bowman *et al* include:

Match-based preference. The match-based preference $\prec_M$ defines four levels of precision which may be used to select attributes. These levels of precision correspond to the approximation functions: *possible*, where the matching is incomplete and the object may have additional unmatched attributes; *partial*, where the matching was partially completed with no additional attributes; *exact*, where an exact matching was found for multiple objects; and *unique*, where an exact matching was found for a single object. The preference order of these approximation functions can be expressed as:

$$possible \prec_M partial \prec_M exact \prec_M unique$$

Voting preference. The voting preference defines a preference scheme where a matched attribute is taken as a vote for the matched object, with three approximation functions corresponding to voted for, but being in the minority; voted for and receiving the majority of the votes; and receiving all of the votes. The preference of these approximation functions can be formally defined as:

$$also{-}ran \prec_V majority \prec_V unanimous$$

Temporal preference. The temporal preference $\prec_T$ denotes a preferences for attributes that have been recently defined or validated. The temporal preference is often used in distributed databases, where some information is cached, and could possibly be out-of-date, where other information is authoritative, giving:

$$out{-}of{-}date \prec_T cached \prec_T authoritative$$

A selection of the appropriate preferences is combined to produce a name resolution scheme implemented by a *resolution function.* A *resolution function* can be defined as an ordering of client and database preferences and represents the resolution scheme for the naming system. The ordering of the preferences gives an ordering within the resolution function producing an ordering of the possible combinations of results.

Equation 3.1 shows an example resolution function for a simple name resolution scheme where the information in the database is accurate but some is cached, motivating a temporal preference. The client presents the naming scheme with correct information indicating a match-based system without the need for a possible match, with all information guaranteed to be in the database. The preference hierarchy for this function, $\prec_M \triangleleft \prec_R \triangleleft \prec_T$, indicates that a temporal preference is given precedence over a match-based preference. To the client this resolution function indicates that its result has a preference for providing a value in a timely fashion over providing an exact match.

$$\begin{aligned}
\prec_T &: cached \prec_T authoritative \\
\prec_R &: closed \\
\prec_M &: partial \prec_M exact \qquad (3.1)
\end{aligned}$$

The inclusion of the registered preference is due to the guarantee that all information is required in the database. Since there is no information that may not be guaranteed inclusion, there is no need to establish a preference between unguaranteed and guaranteed information, but simply to specify that guaranteed information is a requirement of the system. For example, a database may contain information defining several attributes including a service name and a location, which are guaranteed. A client that specifies a closed preference will only be interested in information that is guaranteed and will ignore all other information.

Each possible combination of the preferences can result from the resolution function. The set of possible combinations for this example consists of:

$$\begin{array}{rl} \{cached, closed, exact\} & \{authoritative, closed, partial\} \\ \{cached, closed, partial\} & \{authoritative, closed, exact\} \end{array}$$

These possibilities are ordered in the order of the individual preference of each approximation function and the ordering of the preferences themselves, this gives an ordering of the possibilities in this example of:

$$\begin{array}{rcl} \{cached, closed, partial\} & \prec & \{cached, closed, exact\} \\ & \prec & \{authoritative, closed, partial\} \\ & \prec & \{authoritative, closed, exact\} \end{array}$$

A resolution function for this example will return the first non empty set of objects found by applying the preference matchings in their order of preference. If the resolution function completes without producing a non empty set then resolution fails.

Simple name resolution schemes that require an exact match of a single attribute (being the name of an object) can also be described in this model. Bowman *et al* introduce a special resolution function *cons* which requires an exact match against an unambiguous name entry and hence has the following features:

$$\begin{array}{rcl} \prec_M & : & exact \\ \prec_P & : & unambiguous \\ \prec_T & : & cached \prec_T authoritative \end{array}$$

There features have the preference order $\prec_T \lhd \prec_P \lhd \prec_M$ with matching preference pairings:

$$\{exact, unambiguous, cached\} \prec \{exact, unambiguous, authoritative\}$$

Bowman *et al* present more preferences that may be suitable for naming systems in [26] and define formal methods for reasoning about resolution functions. Bowman *et al* formally define the resolution function and also functions that implement the client and database preferences defined in this section. These formal definitions are described in the next section.

3.3.3 Formalisation of the Naming Models

Bowman *et al* define a formal model for defining client and database preferences using *approximation functions*. These approximation functions represent a model for defining a preference ordering and are used by the resolution function to construct an ordered set of results from a resolution. Within these definitions, the set of objects known to the object system is defined as **O**, the set of names known to the object system is defined as **N**, the set of attributes known to the object system is defined as **A** and the database containing all known bindings of names and objects is defined as **D**.

A *client approximation function* is used to construct a set of attributes that accurately describes the objects sought by the client according to the resolution scheme defined. A client may have several client approximation functions that are ordered by preference, each of which always returns a subset of those attributes available, *i.e.* those that match the preference(s) specified. A client approximation function is defined over the domain of attributes **A** where 2^A represents the power set, or the set of subsets of **A**. A client approximation function is defined formally as:

Definition 3.1 (Client Approximation Function). A client approximation function over a domain of attributes **A** is a function $f : 2^A \rightarrow 2^A$ that is monotonic increasing; that is, for the sets of attributes $\mathbf{a_i}$ and $\mathbf{a_j}$, $\mathbf{a_i} \subseteq \mathbf{a_j}$ implies that $f(\mathbf{a_i}) \subseteq f(\mathbf{a_j})$.

A client approximation function can be understood as a function that takes a set of attributes that the client requires and returns the set of objects that match those attributes. The approximation function obeys the rule that for a given set of attributes $\mathbf{a_i}$ that is a subset of a set of attributes $\mathbf{a_j}$, then the set of objects returned by matching $\mathbf{a_i}$ to the database of objects will be a subset of the set of objects returned by matching $\mathbf{a_j}$. This means that no attribute restricts another and, by adding an attribute to an attribute set, previously included objects can not be removed.

Bowman *et al* define database approximation functions that can be used to match attributes to objects with a defined precision at the database level. The database approximation function is used to match a set of attributes or names against a database of bindings between names and objects known to the system. The database approximation function is defined as:

Definition 3.2 (Database Approximation Function). A function defining database approximation over a domain of attributes **A** and a domain of objects **O** is a function $m : 2^A \times 2^O \rightarrow 2^O$ that is a contradiction on its second argument; that is, for any set of attributes **a** and any set of objects **o**, $m(\mathbf{a}, \mathbf{o}) \subseteq \mathbf{o}$.

The definition of the database approximation function specifies that it never adds objects to its database; *i.e.* for a resolution query for a set of attributes **a**, the set of objects returned by the approximation function must always return a subset of the objects held within the database. The function also specifies that as the number of objects held in the database increases, the number of objects returned by a database approximation function increases except for the unique case, where only one object or zero is ever returned.

A resolution function, as described in Section 3.3.2, returns the first non-empty set of objects found by applying the approximation functions for the preferences defined in the resolution model in their order of preference. Given that the set of approximations of $\prec_j$ is given by π_j, a resolution function can be formally defined as:

Definition 3.3 (Resolution Function). A resolution function can be defined by an induced preference order $\prec$ on a set of approximation functions $\Pi = \{\pi_1, \pi_2, ..., \pi_n\}$ as $\rho : 2^A \times 2^O \to 2^O$ over a domain of attributes $\mathbf{A}$ and a domain of objects $\mathbf{O}$ for any set of attributes $\mathbf{a}$ and any set of objects $\mathbf{o}$ by $\rho(\mathbf{a}, \mathbf{o}) = \pi_i(\mathbf{a}, \mathbf{o})$, $i \in 1..n$, where

1. $\pi_i(\mathbf{a}, \mathbf{o}) \neq \emptyset$, if $i < n$, and
2. $\forall \pi_j \in \Pi[\pi_i \prec \pi_j$ implies that $\pi_n(\mathbf{a}, \mathbf{o}) = \emptyset]$.

The first qualification of the resolution function defines that only non-empty sets will be returned; the second qualification defines that when all approximation functions are applied no objects are returned and, hence, resolution fails.

A resolution function takes a ordered set of approximation function, such as:

$$\prec_M \lhd \prec_T \lhd \prec_R$$

where $\prec_R$ is of highest preference, and applies each approximation function in turn until each approximation function has completed. The definition of a resolution function implies that, for the resolution function to succeed, each approximation function returns a set of objects and eventually, when all preferences are applied, all objects can be removed, *i.e.* each attribute must be able to be selected or deselected using an approximation function.

The formal model of resolution defined by Bowman *et al* also includes methods for verifying the soundness and completeness of a resolution function. These methods can be applied to $\triangle$, where $\triangle$ represents the domain of all attributes $\mathbf{a}$ and objects $\mathbf{o}$ that are in the current domain. The definitions of these properties assume the existence of an *oracle* function that returns exactly the objects that the client wanted. The properties of soundness and completeness can be defined as:

Definition 3.4 (Soundness). Given a set of attributes $\mathbf{a}$ and a set of objects $\mathbf{o}$, a name resolution function ρ is defined as *sound* for $(\mathbf{a}, \mathbf{o})$ in $\triangle$ if, and only if, $\rho(\mathbf{a}, \mathbf{o}) \subseteq oracle(\mathbf{a}, \mathbf{o})$.

Definition 3.5 (Completeness). Given a set of attributes $\mathbf{a}$ and a set of objects $\mathbf{o}$, a name resolution function ρ is defined as *complete* for $(\mathbf{a}, \mathbf{o})$ in $\triangle$ if, and only if, $oracle(\mathbf{a}, \mathbf{o}) \subseteq \rho(\mathbf{a}, \mathbf{o})$.

Soundness is defined informally as a property of a resolution function such that the resolution function returns only sets that are subsets of the correct result set, *i.e.* it does not return any incorrect objects but may not return the complete set of correct objects.

This is true for each ordering of preferences matched within the resolution function, given that the object set returned is the first in a preferential ordering.

Completeness is defined informally as a property of a resolution function that returns only sets for which the correct set is a superset. This means that a complete resolution function returns all objects that match including every object that would be a complete match. A complete resolution function is potentially less desirable than a sound function, however it may return objects that although not accurately matching the client's attribute specification may still be useful. An example of a resolution system that is complete is a yellow pages system, where the client is interested in everything that matches certain attributes and may also be interested in objects that match some attributes but conflict with others. This situation occurs in a system where clients' knowledge is not guaranteed to be completely accurate.

The formalism proposed by Bowman *et al* can be used to define a preference order for naming schemes taking into account both the database and client-based resolution schemes and their internal preferential order. A preference-based system is intuitive for a resolution scheme and can be used to define name resolution systems developed prior to this formalism [26].

The model of formal definition developed by Bowman *et al* can be extended to formally define *binding functions* for the attributes defined by Bayerdorffer as the characteristics of name binding. Through this formal definition, the implementation of the name binding system can be separated from the development of the name binding model.

Binding occurs by taking the attributes that a system exhibits and ensuring that new name bindings match the attribute requirements. As an extension to Bayerdorffer's model formal definitions of binding functions for each attribute are defined in this thesis. A complete binding system within a naming system successively applies its binding functions until it either succeeds and completes all functions or until one function fails, upon which the binding fails.

A binding function can be defined as a set of operations and requirements over the set of names $\mathbf{N}$, the set of attributes $\mathbf{A}$ and the database, $\mathbf{D} \in (\mathbf{N}, \mathbf{O})$, that holds all entered names and attributes and matches the set of objects, $\mathbf{O}$ known to the system.

Each of the binding functions that follow are functions of the set of names, in the domain of $\mathbf{N}$, and the set of objects, in the domain of $\mathbf{O}$, to be bound that produce a database of the bindings between the known names and objects. A system that utilises a selection of the binding functions will apply each of the binding functions in turn to the *(name, object)* pair, indicated by (n, o), to be bound. Each function may produce changes to the database, if one of the functions fails then all alterations to the database resulting from this binding are rolled back.

The attribute of mutability can be defined as the attribute expressed by a naming system where a known binding between a name and an object can be altered, with the existing name being rebound to another object. A name system that is mutable can alter the object bindings for a name if the name exists, otherwise the *(name, object)* pair will be added to the database. A statically bound naming system will only add objects to the database if the name does not already exist within the database.

The mutability binding function is a function that can add a binding or change a binding to reference a new object. For a given *(name, object)* pair, if the name is known to the system, then the existing binding for the name is removed and a new binding is added to the system which links the the name to the new object. If the name is unknown, the new binding is added.

Definition 3.6 (Mutability Binding Function). A name binding function supporting mutability over the domain of names **N** and the domain of objects **O** is a function $f_B : N \times O \rightarrow N \times O$, where for a database of bindings $\mathbf{D}$, and a name n_i and an object o_i to be bound:

1. if $\exists n_j \in \mathbf{N}$ where $n_i = n_j$, then:
   (a) $f_B(n_i, o_i) = \mathbf{D} \setminus (n_j, o_j)$, and
   (b) $f_B(n_i, o_i) = \mathbf{D} \times (n_i, o_i)$
2. else $f_B(n_i, o_i) = \mathbf{D} \times (n_i, o_i)$.

The statically bound binding function is a function that can only add a binding if the name to be bound is not already bound to an object. If the name is known to the database, the function results in a *failure*, otherwise the function adds the binding of the name and object to the database.

Definition 3.7 (Statically Bound Binding Function). A name binding function that supports static binding over the domain of names **N** and the domain of objects **O** is a function $f_b : N \times O \rightarrow N \times O$, where for a database of bindings $\mathbf{D}$, and a name n and an object o to be bound:

1. if $n \notin \mathbf{N}$ then $f_b(n, o) = \mathbf{D} \times (n, o)$
2. else $\rightarrow failure$.

An open knowledge name binding system can be defined as a naming system that allows new names to be bound into the database. In an open binding system, the function allows names to be bound regardless of whether it knows about the name: it will add the name to its set of names and the binding to the database. An open knowledge binding function can be defined formally as:

Definition 3.8 (Open Knowledge Binding Function). An open knowledge name binding function over the domain of names **N** and the domain of objects **O** is a function

$f_D : N \times O \rightarrow N \times O$ where for a database of bindings $\mathbf{D}$, and a name n and an object o to be bound, $f_D(n, o) = \mathbf{D} \times (n, o)$.

A closed knowledge naming system only allows objects to be bound if the name is already known to the naming system; the binding function will only add the binding to the database if the name to be bound exists within the set of known names, otherwise the function results in *failure*. This function can be defined formally as:

Definition 3.9 (Closed Knowledge Binding Function). A closed knowledge name binding function over the domain of names $\mathbf{N}$ and the domain of objects $\mathbf{O}$ is a function $f_d : N \times O \rightarrow N \times O$ where for a database of bindings $\mathbf{D}$, and a name n and an object o to be bound:

1. if $\exists n \in \mathbf{N}$, $f_d(n, o) = \mathbf{D} \times (n, o)$
2. else $\rightarrow failure$.

A name binding system that supports multiplicity allows a name to be bound to multiple objects. A name system that supports singularity only allows a name to be bound to a single object. Multiplicity can be defined simply as an extension to the database without any required checks on existing names; singularity requires an existence check on the existing name before potentially adding the *(name, object)* pair to the database.

A multiplicity binding function allows bindings to be entered regardless of whether the name already exists and is bound in the database. The multiplicity binding function has the same effect as the open knowledge binding function in that it allows the binding to proceed, and be entered into the database, with no precondition. These functions are unrestricting binding functions and are used as a foil to their more restrictive orthogonal functions. Multiplicity can be formally defined as:

Definition 3.10 (Multiplicity Binding Function). A multiplicity binding function over the domain of names $\mathbf{N}$ and the domain of objects $\mathbf{O}$ is a function $f_M : N \times O \rightarrow N \times O$ where for a database of bindings $\mathbf{D}$, and a name n and an object o to be bound, $f_M(n, o) = \mathbf{D} \times (n, o)$.

A singularity binding function will only add a binding to the database if the name is not already bound to an object: a name can only be bound to one object at a time. If the name is already bound to an object then the function results in failure. Singularity can be formally defined as:

Definition 3.11 (Singularity Binding Function). A singularity binding function over the domain of names $\mathbf{N}$ and the domain of objects $\mathbf{O}$ is a function $f_m : N \times O \rightarrow N \times O$

where for a database of bindings $\mathbf{D}$, and a name n and an object o to be bound:

1. if $n \notin \mathbf{N}$ then $f_m(n, o) = \mathbf{D} \times (n, o)$
2. else $\rightarrow failure$.

A naming system that supports binding of names to objects in a many-to-one relationship allows aliases to be created for objects. An unaliased system allows only one name to be bound to each object. Such a system requires the object to not have a pre-existing name binding in order to allow binding to complete.

The alias binding function allows a name to be bound to multiple objects and hence does not restrict the name set or object set to be bound. The alias binding function can be formally defined as:

Definition 3.12 (Alias Binding Function). An alias binding function over the domain of names $\mathbf{N}$ and the domain of objects $\mathbf{O}$ is a function $f_A : N \times O \rightarrow N \times O$ where for a database of bindings $\mathbf{D}$, and a name n and an object o to be bound, $f_A(n, o) = \mathbf{D} \times (n, o)$.

The unaliased binding function only adds a binding to the database if the object does not already have a binding to a name; if the object already exists in the database, the functions results in a *failure*. The unaliased binding function can be formally defined as:

Definition 3.13 (Unaliased Binding Function). An unaliased binding function over the domain of names $\mathbf{N}$ and the domain of objects $\mathbf{O}$ is a function $f_a : N \times O \rightarrow N \times O$ where for a database of bindings $\mathbf{D}$, and a name n and an object o to be bound:

1. if $o \notin \mathbf{O}$ then $f_a(n, o) = \mathbf{D} \times (n, o)$
2. else $\rightarrow failure$.

A naming system that has the shared names attribute allows a name to be shared by more than one object at a time, allowing multiple clients to have references to the one object. An interesting feature of this attribute is that in a naming system with private names and the aliased attribute, individual per client names can be established. The shared names binding function can be defined formally as:

Definition 3.14 (Shared Names Binding Function). A shared name binding function over the domain of names $\mathbf{N}$ and the domain of objects $\mathbf{O}$ is a function $f_S : N \times O \rightarrow N \times O$ where for a database of bindings $\mathbf{D}$, and a name n and an object o to be bound, $f_S(n, o) = \mathbf{D} \times (n, o)$.

The attribute of name sharing also has implications on resolution functions defined for the system, *i.e.* when a shared name can be resolved by multiple clients at a time. Name resolution functions can be used in a similar way to binding functions. They can be applied in succession until either the resolution completes and a *(name, object)* pair is found, or one of the resolution functions fails and in turn the resolution itself fails. The name resolution functions defined here can be combined with the resolution functions of Section 3.3.2. A name resolution function for the shared names attribute can be defined as:

Definition 3.15 (Shared Name Resolution Function). A shared name resolution function over the domain of names $\mathbf{N}$ and the range of objects $\mathbf{O}$ is a function $r_S : N \rightarrow O$ where for a name n bound to an object o:

1. if $n \in \mathbf{N}$ then $r_S(n) = o$
2. else $\rightarrow failure$.

A private name binding function allows any binding to be added to the database. Again, it is the corresponding resolution function that is most interesting as names must be restricted to being used by one client at a time. A private name binding function can be defined formally as:

Definition 3.16 (Private Name Binding Function). A private name binding function over the domain of names $\mathbf{N}$ and the domain of objects $\mathbf{O}$ is a function $f_s : N \times O \rightarrow N \times O$ where for a database of bindings $\mathbf{D}$, and a name n and an object o to be bound, $f_s(n, o) = \mathbf{D} \times (n, o)$.

A naming system that supports private name resolution can be accomplished by using two databases, where $(name, object)$ pairs are moved to the known database $\mathbf{D_K}$ when referenced and back again when the reference is complete. Assumptions on this system are that the naming system can always be sure when a reference is being used and when it is not.

If the name is currently in the available database then it is able to be resolved, otherwise the function results in a *failure*. The private name resolution function can be defined formally as:

Definition 3.17 (Private Name Resolution Function). A private name resolution function over the domain of names $\mathbf{N}$ and the range of objects $\mathbf{O}$ is a function $r_s : N \rightarrow O$ where for a database of bindings $\mathbf{D}$, a database of known bindings $\mathbf{D_K}$, and a name n to be resolved to an object o:

1. if $n \in \mathbf{N}$ then:
   (a) $r_S(n) = \mathbf{D} \setminus (n, o)$,
   (b) $r_S(n) = \mathbf{D_K} \times (n, o)$, and
   (c) $r_S(n) = o$
2. else $\rightarrow failure$.

Descriptive naming allow names to be annotated with attributes that define semantic elements or the state of the object; these attributes may change over time. Alternatively a descriptive name may be a single name constructed in part by a dynamic state-based attribute. In these cases, names are viewed as a combination of several parts, each of which can be examined individually to compare attribute values. Databases of attributes can be maintained by the naming system to facilitate comparisons. The naming system must support the dynamic addition and removal of states as named objects transfer between them. The descriptive name binding function can be defined formally as:

Definition 3.18 (Descriptive Name Binding Function). A descriptive name binding function over the domain of states $\mathbf{S}$, the domain of names $\mathbf{N}$ and the domain of objects $\mathbf{O}$ is a function $f_R : S \times N \times O \rightarrow N \times O$ where for a database of bindings $\mathbf{D}$, and a state s, a name n and an object o to be bound: $f_R(s, n, o) = \mathbf{D} \times (s, n, o)$.

Similarly to the shared names attribute, this attribute has implications on resolution as a naming system that supports descriptive names allows a state identifier to be matched as well as the name of the object. The descriptive name resolution function requires a function *state* that takes a state and a name and returns true if the name in question represents the given state. A descriptive name resolution function can be defined formally as:

Definition 3.19 (Descriptive Name Resolution Function). A function that defines descriptive name resolution over the domain of states $\mathbf{S}$, the domain of names $\mathbf{N}$ and the range of objects $\mathbf{O}$ is a function $r_R : S \times N \rightarrow O$ where for a state s and a name n bound to an object o:

1. if $state(s, n)$, $r_R(s, n) = o$
2. else $\rightarrow failure$.

A naming system that supports nondescriptive naming performs binding between the name and object only, ignoring any state information. A nondescriptive name binding function can be defined formally as:

Definition 3.20 (Nondescriptive Name Binding Function). A nondescriptive name binding function over the domain of names $\mathbf{N}$ and the domain of objects $\mathbf{O}$ is a function $f_r : N \times O \rightarrow N \times O$ where for a database of bindings $\mathbf{D}$ and a name n and an object o to be bound, $f_r(n, o) = \mathbf{D} \times (n, o)$.

A naming system that does not allow descriptive naming simply returns the object if known without any state comparison. The resolution function that implements this ignores any state information and resolves the name given to the object. A nondescriptive name resolution function can be formally defined as:

Definition 3.21 (Nondescriptive Name Resolution Function). A name resolution function for nondescriptive naming over the domain of names $\mathbf{N}$ and the range of objects $\mathbf{O}$ is a function $r_r : N \rightarrow O$ where for a name n bound to an object o:

1. if $n \in \mathbf{N}$, $r_r(n) = o$

2. else $\rightarrow failure$.

These formal definitions of the name binding attributes and, in some cases, their name resolution implications enable more detail about possible implementations and assumptions of naming systems with these attributes to be inferred. The set of binding functions that define a particular name binding model formally define the restrictions and requirements of the binding system. The required operations on the name binding database can be derived from the formal binding function definitions.

A complete binding function can be defined as a function that takes a set of binding functions that represent a binding model and applies each binding function in turn to a $(name, object)$ pair to be bound. The complete binding function applies each binding function in the order of their specification in the naming model until either one binding function fails or all complete. Upon completion, the name binding has been successfully added to the database $\mathbf{D}$ that stores the pairs of $(name, object)$ bindings. A complete binding function can be formally defined as:

Definition 3.22 (Complete Binding Function). A complete binding function can be defined by an induced preference order $\prec$ on a set of name binding functions $\Pi = \{\pi_1, \pi_2, ..., \pi_n\}$ as $\rho_b : 2^N \times 2^O \rightarrow 2^N \times 2^O$ over a domain of names $\mathbf{N}$ and a domain of objects $\mathbf{O}$ by $\rho(\mathbf{n}, \mathbf{o}) = \pi_i(\mathbf{N}, \mathbf{O})$, $i \in 1..n$, where:

1. $\pi_i(\mathbf{N}, \mathbf{D}) \neq failure$, if $i < n$, and

2. $\forall \pi_j \in \Pi[\pi_i \prec \pi_j$ implies that $\pi_n(\mathbf{N}, \mathbf{D}) = \perp]$.

The implication of the complete binding function is that, given a name binding model defined by a set of binding functions, each binding function is applied in turn until all are applied or failure occurs. A failure results in the rollback of any changes to the database consequent to previously evaluated binding functions. The complete binding function consisting of all possible binding functions results in the addition to the database of only those bindings that are valid within the naming model denoted by $\perp$. Within an orthogonal set of binding functions only the most restrictive, and hence present within $\perp$, is applied to a binding.

These formal definitions represent a formal model of Bayerdorffer's work, produced by extending the formal models developed by Bowman *et al.* By defining these attributes formally, and through the development of a complete binding function, the implementation of a name binding system defined by these attributes can be performed through the implementation of each of the defined functions. The formal expression of these attributes also indicates the effect of the attribute. In particular, it is seen that the shared names and descriptive names attributes have effect on both name binding and name resolution. Hence, these attributes must form part of both the resolution function and the complete binding function.

These formal definitions are used to guide the implementation of the naming system used within the DISCWorld ORB system. Through the application of the resolution functions and the complete binding functions, the development of a generic naming system can be achieved. This generic naming system can be used to support any naming system defined in terms of the naming models introduced in this thesis.

3.4 Summary

Naming objects in mobile and distributed systems requires name binding and name resolution schemes to allow the system to create name to object bindings and to resolve a name to an object reference. The inclusion of a naming system within a mobile or distributed system provides the system with the ability to track and contact named objects; migrating named objects are able to be identified and contacted regardless of their location if the naming system used provides location transparency and relocation transparency.

The naming scheme used within an object system provides the system with the building blocks for distributed and inter-object communication. The ability to find an object without needing to know its location provides the necessary physical abstractions required in large distributed systems and metasystems. Additionally, the ability to resolve an object using state-based or semantic information allows objects to act as services; clients can obtain a reference to an object without even knowing the appropriate name, but by knowing the *type* of service they require.

Bayerdorffer and Bowman *et al* define classification models for name binding and name resolution respectively. These models attempt to categorise how names are used within object systems and to provide a framework for comparing and analysing naming systems. Bowman *et al* define an extensible framework upon which additional approximation functions and preferences can be easily defined appropriate to the naming schemes requiring classification.

Bowman *et al* formally define functions that represent each resolution approximation function. This technique has been extended to Bayerdorffer's classification model to define formal binding functions that represent each of the attributes. An implication of these formal definitions is the derivation of a complete binding function that is able to apply an unspecified set of binding functions to a naming database. Using a complete binding function and the formal definitions of the binding functions, a naming service can be developed that is capable of operating under the restrictions of any naming model fitting the classification (see Section 5.4).

A classification of the types of name binding and name resolution schemes present in the naming systems of existing mobile and distributed object systems is presented in Chapter 4. This classification is used to highlight areas where the systems are similar, and to highlight areas where these classification models need to be extended to fully support classification for the object systems in question. In Chapter 5 this information is used to describe and define extensions to the name binding and name resolution models that can be used to describe a system that needs to provide relocation transparency for mobile objects and a powerful and flexible resolution model for service selection.

Chapter 4

Classification of Existing Systems

This chapter presents a classification of the name binding and resolution systems of existing mobile and distributed object systems, some of which are described with respect to their location and relocation mechanisms in Chapter 2. Each system is classified according to the classification schemes described in Chapter 3.

The systems are divided into their area of classification, *i.e.* all systems that share a common attribute are presented together. For simplicity, name binding categories are separated from name resolution categories. Some systems appear in both, while others, for which only one area is of interest, appear in only one. This chapter concludes with a summary of the classifications, with the full classification of each system described in turn.

4.1 Name Binding Classification

Name binding systems as classified by Bayerdorffer's [15,16] scheme (described in Chapter 3) take into account how names are bound and used. Additionally, this classification scheme reflects some elements of the contents of a name, such as whether it contains state information.

Each of the systems mentioned in this section is described relative to its support for the attributes defined by Bayerdorffer in his classification model. Each system appears in the section belonging to an attribute that it supports. Within each section, it is then possible to highlight the similarities and differences in the support that each system provides, particularly in the support differences between mobile and distributed object systems as a group.

4.1.1 Mutability

Mutability requires the ability to change the binding between a name and an object: a name that is already present in a binding to one object has that object replaced with another. This form of binding alteration is present usually in two forms: active rebinding and name reuse.

Active rebinding involves replacing a known active object with another, while maintaining any references and active communications between that active object and possible client objects. Name reuse involves rebinding a name to a new object after a time period where it is assumed that the previously bound object does not now exist within the system.

Active rebinding is the form of mutability often seen in distributed object systems, where long lived services need to be replaced due to version updates. This is the form of mutability found in systems such as Java RMI [55, 167], Globus [42, 54, 57, 58], Grapevine [161], Ajents [95] and Regional Directories [5, 6]. Mutability is often restricted in long lived distributed systems, such as CORBA [18, 136, 138], Infospheres [33, 34], DCE [108, 109] and DCOM [27, 155], because of the difficulties in keeping references and the databases of references coherent.

DCOM allows a form of active rebinding in that DCOM monikers can be rebound to different server objects in response to server of node failure. However, when a moniker is rebound it becomes a new name rather than being rebound as the same name.

Legion [73–75] is an example of a distributed object system that does not support mutability. Each Legion system maintains a dynamic class map that records information about each object within the system. The semantics of this class map are write-once, read-many. These semantics restrict the system in its support for mobility in that an object's location can not be removed. Furthermore, the binding between a name and an object can not be replaced through either active rebinding or name reuse.

Aleph [43, 84–86] is an interesting example of a distributed and mobile object system that supports mutability. Each name within Aleph consists of a channel name linking two objects combined with the name of the object to be located. Each name consists of the location of the client, the location of the required object (or a directional pointer to the location) and the name of the required object. These names are rebound whenever a required object migrates.

Mobile object systems often do not provide a universal tracking or location service, hence it can not always be determined whether a name is still in use, *i.e.* whether the mobile object is still alive. Mobile object systems deal with this in one of two ways: by requiring that names are unique throughout the life of the system, or by allowing reuse only after a specified time period. Systems that allow name reuse in the second way include Emerald [21, 91, 102].

The Emerald system supports mobility and naming for fine-grained and course-grained objects. Each object name is unique within the Emerald system, which is its object ID. An object ID is used as part of a reference to an Emerald object along with information about the locality of the reference, whether the object can be accessed remotely and information about potential forwarding pointers or object addresses. Emerald object IDs will eventually be reused after a time-frame, when it is assumed that the object is no longer present within the system.

Name reuse is often restricted in mobile object systems as the system can never be certain of the lifetime of an object. The Emerald system is able to use an expected lifetime, and hence name reuse, by providing additional mechanisms for maintaining name coherency. Distributed registries of cached object IDs and their locations are maintained within the Emerald system; if an object ID can not be resolved correctly then a broadcast protocol can be used to locate new object IDs. This means that object IDs that have been assigned to new objects can be removed or updated within the system after reuse.

MOA [131] is an example of a mobile object system that supports mutability through rebinding. MOA is designed as a service-based distributed mobile object system, with many similarities to CORBA. Objects within MOA can register their name with a name server and, at a later date, remove that name binding. This name can then be reused in a rebinding to a new object. In a similar fashion to Emerald, a MOA system copes with coherency problems by providing alternative methods for name resolution, including broadcast and home location mechanisms.

The Aglets [113–115, 172] system is an example of a mobile object system with multi-level mutability support. The Aglets system supports object naming for mobile objects, agent servers (end points for migration) and contexts (workplaces within an agent server where an Aglet may execute). Contexts are named by a combination of the agent server name, the IP address of the host on which it is resident, the port number at which the agent server can be contacted and a locally unique ID for the context object itself. Context names may be reused within the system, whereas Aglet names and agent server names can not be reused.

The d'Agents [71, 72, 110] system uses a *yellow-pages* registry system to support distributed name management and resolution. A yellow-pages registry exists at each d'Agents location and contains a list of services and other, remote, yellow-pages registries. The names (corresponding to locations) of remote yellow-pages contained within a registry may be references to external registries that no longer exist. Additionally d'Agents does not provide any mechanism for relocation of objects. Due to this there may exist references to objects that contain out-of-date location information; these references will not be valid at future referencing. To prevent names being used as references to incorrect objects, names can not be reused within a system lifetime.

The requirement that names are never reused must be met through the provision of a system that can ensure that each name is globally unique, which is in itself a difficult problem. Other systems that have this requirement include Aglets, Telescript [120, 184], Sumatra [1, 151] and MOLE [14].

4.1.2 Knowledge

Most systems support the concept of a dynamic domain or dynamic *knowledge*, enabling a system to grow by integrating new objects and new services. Accordingly, the name space of such a system must also be able to grow. A static domain system executes with a known set of objects and services: sometimes this form of system is acceptable, for example in a restricted service provision system or in a management system for a parallelised computation.

A dynamic domain requires that an object be able to find out about other objects within the same system that did not exist where the initial object was created or integrated into the system. A static domain system can be statically programmed and does not have to provide dynamic mechanisms to link objects together, such as resource discovery.

A mobile object system has an inherently dynamic domain, as mobile objects enter and migrate through a system at any point. Similarly, most distributed object systems also support a dynamic domain to allow dynamic addition of service objects and growth of knowledge between systems. It is rare to find a mobile or distributed system that restricts its knowledge to a static domain. Examples of systems which exhibit a static domain are statically programmed systems where named objects are variables of the language and hence statically defined and named.

The Aleph system is an example of a distributed object system with a static domain. An Aleph system consists of a set of processing elements (PEs) that must be defined as part of a PE group before system instantiation. Once a PE group has been initialised it can not be altered. Names within Aleph are bound to channels that connect a source and target object. A reference to an object d will, for example, have the name reference at PE A of (A, B) where PE B is the next PE in the path to d. The domain of an Aleph system consists of the set of channels that may be created amongst the PEs in the group.

Another example of a system with a static domain is Regional Directories. Regional Directories is an example of a system where the domain of the system is extremely large and well-defined. Each name is heavily syntax structured and is in the form of an IP address; the form of an IP address has a limited number of combinations which produces a static domain. Regional Directories is effected less by the restrictions of a static domain than it is by its syntax bias (see Chapter 3).

4.1.3 Multiplicity

Multiplicity allows a name to refer to a group of objects. This grouping ability may be used to refer to a group of objects that perform the same service with any member of the group selectable for resolution. The grouping ability may also be used to refer to a replicated grouping, where replicas of a service object all receive the same communications and a reference acts as a reference to the group.

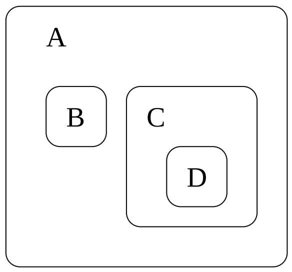


Figure 12: Objects ordered by their creation chain.

It is unusual to find examples of object systems that directly support replication multiplicity. Examples include Sumatra where an object group with the one name can be relocated through the one name reference, and Globe [8, 10, 176–179], where an object handle is mapped to a set of contact addresses which corresponds to the set of addresses for the object replicas. Examples of systems that support service-based multiplicity include Java RMI and Globus. Some mobile object systems (Sumatra, MOLE, Aglets) allow a service type to be part of a name, thus providing service group multiplicity.

MOLE supports a unique form of multiplicity. Objects in MOLE can only have references to objects owned or created by it, external references are not allowed. This produces a grouping mechanism where any of the objects contained within a creation chain can be accessed and migrated using the same name. For example, consider Figure 12, where an object A has created objects B and C, with C then creating object D. A requested migration of object C will migrate both C and D, while a request to migrate A will additionally migrate B, C and D.

MOLE supports another form of multiplicity using abstract locations. An abstract location is a named location that forms an abstraction over a physical location. This provides partial location independence. An abstract location may actually refer to a group of physical locations, with mobile objects distributed over the physical nodes while being resident at the one abstract location. This form of multiplicity is similar to both service group and replication multiplicity in that it supports the selection of one host through an abstract location but can also be used to refer to all hosts.

Infospheres supports a partial form of service group multiplicity. A group of objects can be given a *colour* to define that they are linked in some way. However, these colours can not be used to select or name an object and no ordering or attribute preference can be established within the colours.

4.1.4 Aliasing

Aliasing allows an object to have multiple names. Aliasing is generally found in one of two forms: a predefined or static alias that can not be changed by the object or clients of the object, or a dynamic alias that can be altered by the object during its lifetime. Dynamic aliasing is the form most commonly found in agent systems and distributed object systems, where a service name (or multiple service names) can be defined for an object. These service names can be used by clients and provide an additional abstraction over the object ID. Examples of dynamic aliasing systems include d'Agents, CORBA, and Java RMI.

Allowing an object to have aliases allows the object to be referred to by multiple clients in a private name system (see Section 4.1.5) and also allows an object to respond to any of its compatible superclass names in an object inheritance system. This mechanism is found in the Globus system.

d'Agents uses its service listing mechanism to provide aliasing. The yellow-pages registry system can be used to register multiple service names or aliases that belong to an object. Further to defining aliases, a d'Agents object must specify certain attributes that define the semantic nature of the service.

CORBA, Globus and Java RMI provide similar aliasing for service-based names. CORBA names take the form of IORs [136, 138] which contain detailed information about the object name, version and location. Figure 13 shows an example IOR reference for a CORBA object. Each IOR reference contains a type name, which is used as a repository ID, and optional details that define protocols to be used, contact addresses and the Object Key. The Object Key defines the adapter name (an adapter acts as skeleton) and an optional object name which defines an alias.

When dynamically binding a server into a CORBA system, the binding object can select a single textual name to act as an alias to the object's IOR. These aliases can be used by clients, removing the requirement to know the full IOR reference for an object. Globus provides more simplistic matching against an interface name which acts as an alias for the more complex and detailed object IDs. This interface name is statically defined and derivable from the service object. Java RMI does not allow an object to have explicitly defined aliases, but an object can be registered multiple times under multiple service names.

Examples of static aliasing systems include MOLE, Sumatra and MOA. The same mechanism used to provide multiplicity in MOLE is also used to provide aliasing, where an object can be accessed using any of the names that correspond to objects further up the

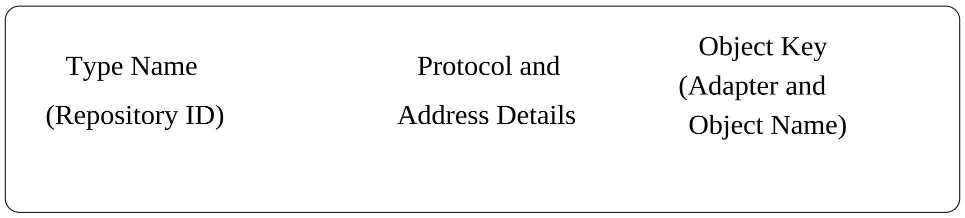


Figure 13: Detail of a CORBA IOR reference.

creation chain. For example consider Figure 12 again, Object D can then be accessed, and be migrated, using any of the following names: D, C or A.

MOA provides a restricted form of aliasing. It allows the derivation of an alternative, location dependent, name from the object's location independent name. This alternative name coincides with the home address of the object which is specified in object creation; this name can then be used to attempt resolution and location of the object.

Sumatra provides a more general form of static aliasing in that it allows the creator of an object to specify a set of aliases at creation time. These aliases are set for the lifetime of the object and can not be altered or added to. In a Sumatra system, each object has a globally unique name. To guarantee that there will be no conflict between an alias provided by the creator of an object, the Sumatra system restricts aliases to being locally unique. Accordingly, aliases can only be used as names by local objects. When a Sumatra object migrates, it can then use its aliases as local names at the new place of execution, but any external references using aliases at the previous site become invalid[6].

Infospheres provides a form of aliasing that is more coherent with Bayerdorffer's view of aliasing. An object may have multiple names, with each name referring to a different *inbox* that acts as a portal to the object. Each inbox is responsible for queuing its own requests and for managing its own communications.

Some aliasing systems restrict their aliasing to subsets of objects within the system, dependent on the object type. For example, MOA provides naming of MOA agents, locations and name servers, while only providing aliasing for MOA agents. This is a common restriction and one that is found in both static and dynamic aliasing systems.

4.1.5 Name Sharing

Name sharing is the ability for multiple clients to share the same name, hence if a system is to support name sharing then a name must contain no specific information about the client side of the communication.

d'Agents is a system where names can be shared; a client reference to a d'Agent object contains the following information: a place name and an object name to distinguish between multiple objects that may be resident at a place, with no reference to the client object. This form of name sharing is common within mobile and distributed object systems [14, 34, 71, 109]. Location independent naming systems are also name sharing systems. One positive side-effect of shared names, as seen in Obliq [29, 30, 94], is that references can be transmitted and copies freely transmitted to anywhere within the network.

Aleph is one of the few systems where names are private. Each name within Aleph consists of a channel map between the client object and the referenced mobile object. Each

[6]The forwarding mechanism described in Chapter 2 can be used in the cases of both local aliases and the object name as they will all reference the same forwarding object left behind by the mobile object.

name is dependent on the location of the client and hence each reference held by a client must be unique (assuming each client object has a unique location, described by a processor element identifier).

Infospheres is an interesting example in the case of name sharing. Infospheres provides for name sharing but also allows an object to prohibit it. It is possible to configure an object to have a single inbox and to only accept one communication or connection at a time using that inbox. Ideally, a system like Infospheres would fit into both the naming sharing and private naming categories.

4.1.6 Descriptive Names

Bayerdorffer defines a descriptive name as a name that contains information that may be used to give information about the current state of the referenced object. Descriptive naming is uncommon in the majority of mobile and distributed object systems in the form described by Bayerdorffer. Examples of descriptive naming of this form include names of entries in Ada [139] tasks, where a name is restricted depending on the ability to enter a task method. An alternative and more widely used definition of descriptive naming is a description of the semantic attributes of an object. These attributes may dynamically change during the lifetime of an object, but do not necessarily describe the state of the object. The category is termed semantic descriptive naming.

Descriptive naming can be easily confused with dynamic aliasing in that the ability to dynamically create an alternative name also provides the ability to create a state-based name. The difference is that it is the initial name that must be able to contain state information.

Examples of descriptive naming in mobile and distributed object systems include MOLE, Aleph and Globus. Names in MOLE are immutable to external objects but do contain some state information, for example names contain the following information:

- a dynamic counter (incremented at each new ID creation),
- a crash counter (incremented at each system restart),
- an IPv6 address of the creation location's physical address, and
- the port number of creation.

This information contains dynamic counter information which can be changed by the system to reflect partial system and object state.

Aleph provides a more traditional (as defined by Bayerdorffer) form of descriptive naming. Aleph provides mutually exclusive access to distributed shared data objects and restricts access to objects based on their current access state. Descriptive naming is provided through this restriction, in that a name is valid and resolvable only when the object is available for exclusive access.

Although descriptive naming is unusual in distributed and mobile object systems, some middleware systems such as CORBA, Globus and Java RMI have chosen to support attribute-based resolution and descriptive naming through optional facilities. The CORBA specification outlines a trading service [137], which uses information provided by the naming service and through the registration process. This trading service can be used to extend a CORBA system from a single level directory to a two-level directory structure. The idea of an integrated trader system or federated traders has also been proposed, allowing the possibility for n-level CORBA systems. A trader can be used to provide state-based descriptive naming and also dynamic semantic naming.

Java RMI and Globus use extensions of the Lightweight Directory Access Protocol [186] (LDAP) to provide attribute-based name resolution and binding. DCE also provides mechanisms to include a trader based on the Open Distributed Processing (ODP) trading function developed as part of the Basic Reference Model of Open Distributed Processing (RM-ODP) [101][7]. These mechanisms can be used to support static semantic descriptive naming but not dynamic state-based descriptive naming.

4.1.7 Summary

The attributes chosen to define name binding by Bayerdorffer cover many areas of name binding within mobile and distributed object systems. However, as has been shown, in many cases the attributes chosen are too general and refinement is needed to correctly classify existing systems.

Many existing systems have support for complex naming models with expressive binding. Some are very general and allow a name to define, primarily, what is a service type. Each individual object is recognised purely as an example or instance of the service. Other systems treat each object's identity as unique and do not allow equivalency relationships to be set up between objects. Even with the differences shown in the name binding models of these systems, they can be classified according to their support for certain attributes.

4.2 Name Resolution Classification

To fully classify a naming system both the name binding and resolution models must be examined. Bowman *et al* [26] define a framework for classifying attribute-based name resolution systems. Bowman *et al* define two categories within their classification: client approximation and database approximation functions. Client approximation functions

[7]In fact, work on DCE traders preceded CORBA's development and has had great influence on the development of trader systems.

define the client's preference for ordering of attributes; database approximation functions define a preference for the correctness of a result. For example, a database approximation function may define a preference for an exact match over a possible match while another database approximation function may define a preference for an authoritative response over a cached response (see Section 3.3.2).

While the name resolution systems of most mobile and distributed object systems are not attribute-based, many do present some of the approximation functions described by Bowman *et al* such as a preference for cached over authoritative data. Many service-based distributed object systems provide limited attribute-based resolution with the only attribute available for comparison being the service name. This ability also motivates the inclusion of approximation functions that specify a match-based preference or a temporal preference.

This section examines the support for the approximation functions and preferences, defined in Section 3.3.2, as shown by existing mobile and distributed object systems.

4.2.1 Registered Preference

A registered preference displays a preference for attributes that are guaranteed to exist within the database over those that are not. Any object that has the guaranteed attribute must have that attribute registered within the database. This can be expressed as a preference for a service name or location identifier which may be required in a registration over optional attribute specification such as the return type of a method signature.

Distributed object systems that include a registered preference as part of their name resolution model include DCE, Globus, Infospheres and Legion. Many of these systems are LDAP-based and hence have similar name resolution schemes. Each object within these systems is required to have some form of object identifier registered. DCE requires an interface name, location identifier and entry point (port number) to be registered for each of its services. These elements form part of the required object handle that acts as a reference to a service object. Globus and Legion additionally have similar attributes, such as a service name, that must be entered for each object with the attributes. Infospheres requires that a home location and a service name be entered for each object.

Restricted preferences, where a preference is defined as a subset of a preference defined by Bowman *et al*, are present in the classification of several mobile and distributed object systems. A restricted preference is defined as a subset of the preference as defined by Bowman *et al*. NetSolve [31, 32] is an example of a distributed object system that has a restricted registered preference. All of the attributes within a NetSolve system are guaranteed and, hence, NetSolve has $\prec_R$: *closed* as a restricted preference.

Although CORBA has a similar trader-based system, it only provides partial support for the registered preference. Attributes that are present in each object are not required to be in any of the naming services provided. For example, each CORBA object has a class type and an IOR. This information is not required for each object within a CORBA system as it is not compulsory to register an object with either the implementation repository or the trading service. However, for each object that is included in a naming database, these attributes are guaranteed.

Grapevine is an example of a mobile object system that supports the registered preference. A Grapevine system stores the address of multiple inboxes for each Grapevine client. A client is required to have a single primary inbox and may have multiple secondary inboxes. When name resolution is performed, a Grapevine name server attempts to resolve to the most appropriate inbox, with a preference placed on the primary inbox and also on the locality of the inbox. This implementation of a registered preference is unique to Grapevine and is an example of a specific usage. Most mobile and distributed object systems that support the registered preference allow the preference to be given to any attribute. This is especially true of systems that use an LDAP-based system, where the support for attribute classification (*i.e.* guaranteed, optional) is separate from the rest of the naming system.

Mobile object systems that support the registered preference include Aglets, Emerald and Telescript. Aglets does not support an attribute-based resolution system, however resolution can be performed through the specification of an Aglet identifier. This identifier is guaranteed to be registered for each Aglet within the system. Similarly, Telescript requires that all attributes be guaranteed in its simple name resolution model. Each object has an attribute defining its type which is the only attribute to be searched upon within the Telescript system. This attribute is guaranteed to be present for each Telescript object. Emerald requires that each process have its unique process ID stored within the access table stored on each processor. These systems have the same restricted registered preference as shown by NetSolve.

Each mobile process system examined in Chapter 2 exhibits the registered preference. Each system requires that each process have its process identifier recorded in some process or access table, in a similar model to that of Emerald.

4.2.2 Mutability Preference

The mutability preference places a preference on attributes that are always defined for an object over those that may change and hence may not always be valid attributes for the object. This preference order is preferable in naming systems that also exhibit state-based descriptive naming or private names. In these systems the mutability preference allows resolution based on the dynamically changing characteristics of the name, such as the current state or the availability.

This preference is not exhibited in many of the distributed object systems examined in this thesis as it is a preference most commonly found in state-based systems such as concurrent systems or systems that support mutually exclusive access. This is unfortunate as the selection of an object with a preference for consistently defined information over dynamically changeable information can be very useful in a mobile or distributed object system.

Some mobile and distributed object systems support a variation of the mutability preference in that they have a preference for dynamically defined information over statically defined information. For example, the name resolution system used within Globus is based on LDAP with extensions to support a time-to-live specification. A time-to-live attribute defines the time period in which the entries information is valid (see Section 4.2.7 for more uses of the time-to-live attribute). This requirement defines a resolution preference for information that is still within its time-to-live period over information that is outside of that period.

Mobile object systems that use a distributed searching mechanism of some kind also display this variation of the mutability preference. In such a system, the location of an object is an implicit attribute that is changed as the object migrates. This attribute is part of a preference as local objects are examined, and may be selected, before remote objects. This issue is examined further in Chapter 5.

4.2.3 Precision Preference

A precision preference is a preference for attributes that are defined for at most one object over those attributes that may be defined for many objects. This preference is present in the name resolution systems of object systems that allow resolution based on a unique identifier and additionally support immutability. The requirement for immutability is implied because throughout the lifetime of the system the identifier can only be used to access a single object. A mutable system will allow the identifier to be reused dependent on the lifetime of the object rather than the system. The requirement for mutability rules out most mobile process systems as process IDs assigned to process objects are eventually reused.

Examples of distributed object systems that support a precision preference include CORBA, DCE, DCOM, Globe, Legion and Infospheres. Each of these systems allow the resolution of a name based on an attribute that is defined for at most one object. For example, the attribute *URL : http://www.infospheres.root/obj1* represents the name of an infospheres object. This name is unique, meaning that it is only defined as an attribute for at most one object. Resolution based on this attribute takes advantage of the precision preference.

Examples of mobile object systems that support a precision preference include d'Agents, Aglets, MOLE, Sumatra and MOA. These systems all have unique identifiers that are immutable and provide resolution mechanisms where the identifier can be used as a search attribute. As the lifetime of an object in a mobile object system is difficult to monitor, immutability and name uniqueness are a common attribute combination in mobile object systems.

4.2.4 Yellow-pages Preference

The yellow-pages preference places a preference on attributes that are required for each object registration over those that are optional. This preference is similar to the registered preference; the difference between these preferences is that when an attribute is guaranteed it is required for all objects that exhibit that attribute, when an attribute is mandatory it is required for each object in the system.

Distributed object systems that use a trader service or an LDAP-based naming service exhibit this preference. These systems often provide the yellow-pages preference with the ability to define optional and mandatory specification as a generic service that can be applied to any attribute. LDAP provides the ability to assign an attribute of a class to be optional or mandatory. Resolution for these objects must include the attribute specification of mandatory attributes. Optional attributes are used to narrow resolution further, for example to select an appropriate server where service name multiplicity is in use. Globus is an example of a system that provides generic support for mandatory and optional attributes and also defines specific attributes that are mandatory. Globus requires that each object register a service name as well as a time-to-live attribute.

MOLE is an example of a mobile object system that supports a similar generic system for providing the yellow-pages preference. MOLE objects can be assigned badges. Badges represent a mechanism for annotating an object with additional information such as aliases, semantic descriptions and state information. These badges can be classified as either an optional or a mandatory part of resolution and hence provide the yellow-pages preference.

The registered preference, which is similar to the yellow-pages preference, differs in that it is generally used for specific attributes such as a service name or location. This is interesting as the semantics of each preference indicate that their implementation or usage should be swapped. A yellow-pages preference is used as a preference for assigning mandatory attributes on a class by class basis, whereas a registered preference is used to guarantee attributes for what is generally all classes of objects within the system.

Many distributed object systems state that they exhibit a yellow-pages-based searching mechanism, whereby they mean that they provide an attribute-based searching mechanism, some of which allow dynamic specification of attributes. This form of searching mechanism is not equivalent to a yellow-pages preference.

4.2.5 Match-based Preference

A match-based preference is the first preference specified for the database approximation function. A match-based preference places a preference on those objects that match identically to the attribute specification over those that only exhibit a partial or possible match. The match-based preference provides an approximation function to describe systems where a single object that matches identically is preferable to all objects that match identically. This preference describes systems that can complete resolution after locating a single matching object.

A match-based preference may exist in systems where a unique selection is not always possible. In these systems a subset of the approximation functions is included that represents the preferences exhibited. For example, the name resolution system of d'Agents includes a match-based preference with the restriction $\prec_M$: *partial* $\prec_M$ *exact*. A unique match is not preferred as the name server can resolve a server name to multiple matching objects. For this reason a unique preference does not often appear in distributed object systems where service selection tends towards producing a complete result rather than a sound result.

CORBA is an example of a distributed object system that includes the unique factor in its preferences. A CORBA system operating with a trader permits a client to request a maximum number of results be returned from an attempted resolution. This number can be set to one, indicating that the client has a preference for a unique match rather than a set of objects.

Many different restrictions within this preference exist in the mobile and distributed object systems studied in this thesis. A possible match is not often found as it is defined as a match where some attributes are matched, while others conflict.

The restricted preference $\prec_M$: *partial* $\prec_M$ *exact* is found in many systems including CORBA and DCE (with trader systems), Java RMI (with LDAP), Globus and d'Agents. This restriction indicates that resolution may return a set of objects of which some are partial matches and some are exact matches. Partial matches must have at least some of the attributes; exact matches must have all of the attributes.

The restricted preference $\prec_M$: *exact* is found in systems where only an exact match is resolved. Examples of these systems are Aleph, NetSolve, Grapevine, Ajents, Sumatra, MOA and Emerald. This preference is commonly matched with the precision preference in systems that do not support a strong attribute-based resolution mechanism. This covers most mobile object systems.

4.2.6 Voting Preference

A voting preference is a preference for objects that are selected unanimously over those that are only partially selected. This translates to a preference for objects that matched all attributes over those that only partially matched the attributes.

The difference between a match-based preference and a voting preference is that a match-based preference places a preference on an identical matching (*i.e.* a unique match is one where all of the attributes in the client specification are matched and the object has no additional attributes) while a voting preference places a preference on matching all of the attributes but does not care about additional attributes that the object may have.

A voting preference is exhibited by systems that allow attribute-based searching and permit the client to specify a certain number of matches to be returned or orders the matches in the order of objects that matched most attributes over those objects that has only a partial matching. Examples of such systems are CORBA and DCE. In CORBA systems the ability to specify how many results should be returned from a resolution attempt and the potential to order solutions according to how many attributes matched provides this preference.

A voting preference is also present in systems where replicated name servers concurrently attempt resolution. In the case where all name servers select the same object, a unanimous decision is reached. When most of the name servers select the same object it is a majority decision, and when few of the name servers select the same object the object is in the also-ran category.

4.2.7 Temporal Preference

A temporal preference is a preference associated with the timeliness of information. A temporal preference places a preference on attributes that are authoritative, *i.e.* the name server has obtained the information first-hand and is not relying on secondary sources. Alternatives are cached information, where information has been provided by an external source, and out-of-date information. Out-of-date information relies on a time-to-live system, where cached information is given a time-frame during which the information is to be considered valid. After the time-frame has elapsed, the cached information is considered out-of-date. Globus is an example of a distributed object system that uses the temporal preference in this way.

This form of preference is present in many mobile and distributed object systems where a form of distributed name server or registry is used. Distributed registry systems that are not replicated or centralised often contain cached, and potentially out-of-date, information about the locations of other name servers, or attributes of objects resident within the system. A temporal preference within this form of system places a preference for information from nodes at which the information is authoritative. For example, a temporal preference within a d'Agents system places a preference on information contained within the local yellow-pages server, and not information contained by remote yellow-pages servers. This preference exists because the locations of remote yellow-pages servers are cached and are not authoritative. Example systems include DCE, DCOM, Aleph, Aglets, Globus, NetSolve, Infospheres and MOLE.

Another example of the usage of this preference is in mobile object systems that use the forwarding location model to provide relocation transparency. Each forwarding object contains cached information about the last known location of the mobile object which may be out-of-date if there is a break in the chain or the mobile object fails. Examples of such systems include MOA, Emerald, Charlotte [3, 4], DEMOS/MP [149] and Sumatra.

4.2.8 Summary

The framework for classification of name resolution systems developed by Bowman *et al* is an extensive and flexible framework. Classifications are not limited by the existing preferences; these preferences act as examples to motivate the development for further preferences and approximation functions.

The classification of existing systems as described in this section shows that although there are vast differences in how the systems support the preference ordering, the classification is detailed enough to encompass all systems. This classification has also shown where additional preferences need to be developed to completely classify existing mobile and distributed object systems (see Chapter 5).

4.3 Classifications

Each system as described both in this chapter and in Chapter 2 has a distinct naming system. Sections 4.1 and 4.2 have described the support for each attribute and preference as found in existing mobile and distributed object systems. Each of the attributes and preferences can be realised in many different ways dependent on the needs of the system in question. As has been seen in the above discussion, the classification is often very broad and does not fully inform the reader of the exact nature of the name binding or name resolution system. Furthermore, many systems have features of their naming models that are not classified by the defined classification models.

This section presents a summary of the naming systems for the example mobile and distributed object systems used throughout this chapter. This summary includes the name binding and resolution models for each system and additional features in their naming models that are outside of the classification.

4.3.1 Mobile Process Systems

The name binding of mobile process systems is typically in the category **BDmaSr**. Each process has a unique identifier which will eventually be reused. The dynamic nature of an operating system, where processes are created and removed at any point, indicates a dynamic domain. Names can be shared by multiple clients.

The name resolution model for mobile process systems is typically very similar to a *cons* model, with the addition of a registered preference. The process ID for each process is guaranteed to be registered in the access tables on each processor. The name resolution model is:

$$\begin{aligned} \prec_R &: \textit{open} \prec_R \textit{closed} \\ \prec_M &: \textit{exact} \\ \prec_P &: \textit{unambiguous} \\ \prec_T &: \textit{cached} \prec_T \textit{authoritative} \end{aligned}$$

4.3.2 Mobile Object Systems

Aglets

Aglets is a Java-based mobile object system. The Aglets system includes a programming library for implementing autonomous and interactive agents over a Java system. An Aglet is a mobile object that can migrate between Aglet servers. Once resident at an Aglet server, each Aglet executes within a named context. Each Aglet has a unique name which is immutable, however context names are mutable, providing multi-level mutability.

The name binding model for Aglets is in the category **bDmaSr**. Aglets supports service-based multiplicity but not replication multiplicity. This a very simple naming model that allows a dynamic domain and sharing of names. No attribute specification is supported in either the name resolution or binding model.

Resolution in an Aglets system is performed using two mechanisms. The first mechanism is to use the Aglet's identifier and its location to request a binding and create a new proxy. The second mechanism is to search the local Aglet server's listing of cached proxies and to use one already in existence. The name resolution model for Aglets is:

$$\begin{aligned} \prec_R &: \textit{open} \prec_R \textit{closed} \\ \prec_M &: \textit{exact} \\ \prec_P &: \textit{unambiguous} \\ \prec_T &: \textit{cached} \prec_T \textit{authoritative} \end{aligned}$$

Each Aglet is assumed to have registered its Aglet identifier with the system, guaranteeing the presence of this attribute. Each Aglet identifier has a single match to an object, hence it is an unambiguous match with an exact result. As the proxies stored by an Aglet server are cached, the temporal preference is also included.

Ajents

Ajents is a Java-based mobile object system, that uses Java RMI as its means of communication and resource discovery. The name binding model of Ajents exists within the

BDmASr category. Each Ajent has a unique name that is mutable. Aliases can be defined by registering the Ajent with the RMI system multiple times. The form of mutability supported by Ajents is active rebinding.

The name resolution model for Ajents maps a unique object identifier to its matching Ajent. The name resolution model for Ajents is:

$$\begin{aligned} \prec_P &: \textit{unambiguous} \\ \prec_M &: \textit{exact} \end{aligned}$$

d'Agents

d'Agents is a Tcl-based mobile object system. The name binding model for d'Agents is in the category **bDmASr**. Each mobile object has a unique name that is immutable and allows aliasing using dynamic service name aliasing.

The name resolution model within d'Agents uses distributed yellow-pages servers where some information is cached. Resolution is service-based, hence requiring an exact, but not unique, match. The name resolution model is:

$$\begin{aligned} \prec_Y &: optional \prec_Y mandatory \\ \prec_M &: partial \prec_M exact \\ \prec_T &: out-of-date \prec_T cached \prec_T authoritative \end{aligned}$$

Emerald

Emerald is a mobile object system that supports fine-grained data object and coarse-grained process object migration. The name binding model for Emerald is in the category **BDmASr**. The form of mutability supported is a name reuse model.

The name resolution model for Emerald is similar to a *cons* model (see Section 3.3.2). There is an exact and unambiguous matching between an object ID and its object. Using forwarding processes as the relocation mechanism introduces cached and potentially out-of-date information into an Emerald system. The name resolution model has the following preferences:

$$\begin{aligned} \prec_M &: exact \\ \prec_P &: unambiguous \\ \prec_T &: out-of-date \prec_T cached \prec_T authoritative \end{aligned}$$

MOA

MOA is a mobile object system that supports multiple relocation mechanisms. Each MOA object is uniquely named. The name binding model of MOA is in the category **BDmASr**. The form of mutability supported is active rebinding. MOA supports a restricted form of aliasing: a single static alias can be defined for each mobile object.

The name resolution scheme for MOA supports a simple, non attribute-based scheme that provides exact matching in a similar way to a *cons* model. One of the multiple relocation mechanisms supported in MOA is the forwarding location model. The use of this model, and the use of a distributed name server, introduces cached and potentially out-of-date information into the system. The name resolution model has the following preferences:

$$\begin{aligned} \prec_M &: \; exact \\ \prec_P &: \; ambiguous \\ \prec_T &: \; out-of-date \prec_T cached \prec_T authoritative \end{aligned}$$

MOLE

MOLE is a Java-based mobile object system that supports autonomous migration and attribute-based resolution. The name binding model of MOLE is in the category **bDmASR**, and is unusual as it is a name binding model that provides state-based descriptive naming. The form of multiplicity supported is service group multiplicity. MOLE objects can be annotated with *badges* that can be used to represent dynamic aliases.

The primary purpose of the badge mechanism is to support dynamic attribute specification where each attribute can be classified as either an optional or mandatory part of resolution. The resolution model used within MOLE has the following preferences:

$$\begin{aligned} \prec_Y &: \; optional \prec_Y mandatory \\ \prec_M &: \; partial \prec_M exact \prec_M unique \\ \prec_T &: \; out-of-date \prec_T cached \end{aligned}$$

Obliq

Obliq is a distributed object system that can be used to support mobility. Obliq supports dynamic aliasing through its reference translation mechanism; as objects are migrated, local references are translated to aliases that act as remote references. The name binding model for Obliq is **bDmASr**.

Each Obliq object is given a unique identifier that provides a unique matching ability. A temporal preference exists as the centralised name server used to maintain the mappings between object identifiers and object locations contains cached and potentially out-of-date information. The use of forwarding aliases as a relocation mechanism also introduces the temporal preference. The name resolution model for Obliq is:

$$\begin{aligned} \prec_M &: \mathit{exact} \\ \prec_P &: \mathit{unambiguous} \\ \prec_T &: \mathit{out{-}of{-}date} \prec_T \mathit{cached} \prec_T \mathit{authoritative} \end{aligned}$$

Sumatra

Sumatra is a Java-based mobile object system used to monitor resources. Sumatra objects have globally unique names and a local set of statically defined aliases. Sumatra supports both replication and service group multiplicity. The name binding model for Sumatra is **bDmASr**.

The name resolution of Sumatra is similar to a *cons* model as a unique matching must be performed between the name provided and the reference given. The use of forwarding objects as the relocation mechanism introduces the temporal preference. The name resolution model for Sumatra is:

$$\begin{aligned} \prec_M &: \mathit{exact} \\ \prec_P &: \mathit{unambiguous} \\ \prec_T &: \mathit{authoritative} \end{aligned}$$

Telescript

Telescript is a scripting language designed to support the development of autonomous agents. The name binding model for Telescript enables objects to refer to other classes of objects but is not essentially a name binding model for an object system. Objects themselves are not named and can not be referred to by other objects. Interaction is solely through the actions of a Telescript agent. The name binding model is **bDMaSr**.

Name resolution is performed using an exact matching between the service name and the location of the required service package. All attributes, in the restricted set of attributes allowed, are guaranteed to be registered. The preferences for Telescript are:

$$\begin{aligned} \prec_R &: \mathit{closed} \\ \prec_M &: \mathit{exact} \\ \prec_T &: \mathit{authoritative} \end{aligned}$$

4.3.3 Mobile Host Systems

Mobile IP

Mobile IP is a standard for supporting mobile hosts within a network. Each mobile host has a home IP or a care-of IP address as aliases. These aliases can be dynamically changed in response to node failure. The domain of this form of system is static, as the set of all IP addresses is bounded. The name binding model for Mobile IP is **BdmASr**. As Mobile IP is a standard to be implemented by other systems it does not provide any resolution mechanism.

Regional Directories

Regional Directories is a distributed directory mechanism designed to support mobile hosts. Regional Directories has a static domain as the set of IP addresses, which correspond to names, is bounded. The form of mutability supported is active rebinding. The name binding model is **BdmASr**.

Regional Directories supports a unique directory system that provides forwarding pointers to the location of each object at multiple levels within the directory. There is an exact match in the directory system between the name of an object and its location. The name resolution model for Regional Directories is:

$$\begin{aligned}
\prec_M &: \textit{exact} \\
\prec_P &: \textit{unambiguous} \\
\prec_T &: \textit{cached} \prec_T \textit{authoritative}
\end{aligned}$$

4.3.4 Distributed Object Systems

Aleph

Aleph is a distributed mobile object system that supports mutually exclusive access to shared objects. The name binding model for the Aleph distributed object system is a **BdMasR** name binding model with location dependent names that are globally unique within the object system. Aleph supports a static domain as the objects within a system are specified upon system initialisation. Names within Aleph consist of channel names that link the client and server location with the object required. These names are mutable; the form of mutability supported is active rebinding.

The location of each object is stored at each node along the path to the object, relying on cached information. The name resolution model for Aleph is:

$$\begin{aligned} \prec_Y &: \textit{mandatory} \\ \prec_M &: \textit{exact} \\ \prec_T &: \textit{cached} \prec_T \textit{authoritative} \end{aligned}$$

CORBA

CORBA is a specification for a middleware system that supports distributed client/server communication regardless of the target platform or implementation language of the client or server objects. The name binding model for CORBA allows dynamic aliasing for service objects. With the support of a trader, CORBA provides both state-based descriptive naming and dynamic semantic description. The classification of the name binding model is **bDmASr** (with a trader it is **bDmASR**), however this does not take into account the ability for aliases to have the multiplicity attribute.

Resolution can be through either a unique mapping between an IOR and its reference object, or the generation of an object set matching an attribute specification. Within the attribute specification system the number of required objects can be limited and orderings performed within the set. CORBA partially supports the registered preference as, for each object that is registered, certain attributes are guaranteed. However, it is not a requirement to register each object. The resolution model is:

$$\begin{aligned} \prec_R &: \textit{open} \prec_R \textit{closed} \\ \prec_Y &: \textit{optional} \prec_Y \textit{mandatory} \\ \prec_P &: \textit{ambiguous} \prec_P \textit{unambiguous} \\ \prec_M &: \textit{partial} \prec_M \textit{exact} \\ \prec_T &: out{-}of{-}date \prec_T \textit{authoritative} \end{aligned}$$

DCE

DCE is a middleware layer designed to support distributed client/server communication. The name binding model for DCE is **bDmASr**. The form of descriptive naming supported (through the use of a trader) is static semantic description.

DCE supports unique naming for each server object. Each object is represented by an object handle which contains information that is guaranteed to exist for every object.

The distributed trader service provided in DCE allows the inclusion of cached information representing services within the system. The name resolution model for DCE is:

$$\begin{aligned}
\prec_R &: \ \textit{open} \prec_R \textit{closed} \\
\prec_Y &: \ \textit{optional} \prec_Y \textit{mandatory} \\
\prec_P &: \ \textit{ambiguous} \prec_P \textit{unambiguous} \\
\prec_M &: \ \textit{partial} \prec_M \textit{exact} \\
\prec_T &: \ \textit{cached} \prec_T \textit{authoritative}
\end{aligned}$$

DCOM

DCOM is a middleware layer based on DCE and an extension to COM [127] developed by Microsoft corporation. The name binding system for DCOM is **bDmASr**. DCOM supports a dynamic domain, aliasing (through redirections) and shared names. As DCOM's naming mechanism is so closely based on DCE, the name binding mechanisms are similar and the classification identical.

DCOM objects are uniquely named, with information about each object stored in the DCOM Class Store and the distributed ROT system. This information includes information that is guaranteed to exist for each object. This information is cached, and through version updating, can become out of date. The name resolution model for DCOM is:

$$\begin{aligned}
\prec_R &: \ \textit{open} \prec_R \textit{closed} \\
\prec_P &: \ \textit{ambiguous} \prec_P \textit{unambiguous} \\
\prec_M &: \ \textit{exact} \\
\prec_T &: \ \textit{out}-\textit{of}-\textit{date} \prec_T \textit{cached} \prec_T \textit{authoritative}
\end{aligned}$$

Globe

Globe is a distributed object system designed to support object replication and migration. The name binding system for Globe is **bDMASr**. An object handle, which exists as the unique immutable name for an object, can have a single symbolic alias. Object handles can map to multiple contact addresses, therefore supporting replication multiplicity.

Resolution is performed using unambiguous information that is cached but never out-of-date. The name resolution system for Globe has the following preferences:

$$\begin{aligned}
\prec_P &: \ \textit{unambiguous} \\
\prec_T &: \ \textit{cached} \prec_T \textit{authoritative}
\end{aligned}$$

Globus

Globus is a distributed object system that provides a metacomputing directory service. The Globus name binding model is one of the more sophisticated classified in this thesis due to its rich attribute-based binding mechanism. Globus supports active rebinding, service group multiplicity and static aliasing. Globus also supports static semantic description. The name binding model is **BDmASR**.

Resolution can be performed through either a unique mapping between an object handle and its referenced object, or through a trader service. The trader service supports a time-to-live attribute specification and also provides generic support for optional and mandatory attribute specification. The name resolution model is:

$$\begin{aligned}
\prec_R &: \textit{open} \prec_R \textit{closed}\\
\prec_U &: \textit{static} \prec_U \textit{dynamic}\\
\prec_P &: \textit{ambiguous} \prec_P \textit{unambiguous}\\
\prec_Y &: \textit{optional} \prec_Y \textit{mandatory}\\
\prec_M &: \textit{partial} \prec_P \textit{exact}\\
\prec_T &: out-of-date \prec_T \textit{cached} \prec_T \textit{authoritative}
\end{aligned}$$

Grapevine

Grapevine is a distributed messaging system designed to support a large number of clients and mailboxes. The name binding model used in Grapevine is one of the more highly expressive naming models present in existing distributed systems. Grapevine supports static aliasing and mutability in the form of active rebinding. The form of multiplicity is service group multiplicity. The name binding model is **BDMASr**.

The name resolution model for Grapevine is not attribute-based and is similar to a *cons* scheme. It represents a specific usage of the registered preference that is isolated to the guaranteed registration of primary inboxes. Cached and potentially out-of-date information exists within a Grapevine system because the locations of inboxes are stored in various nodes within the system. The name resolution model has the following preferences:

$$\begin{aligned}
\prec_R &: \textit{open} \prec_R \textit{closed}\\
\prec_P &: \textit{ambiguous}\\
\prec_M &: \textit{exact}\\
\prec_T &: out-of-date \prec_T \textit{cached} \prec_T \textit{authoritative}
\end{aligned}$$

Infospheres

Infospheres is a distributed, web-based, object system that utilises URLs as its naming mechanism. The name binding model for Infospheres is interesting because it exhibits dual attributes for name sharing. This name model allows name sharing but can also prohibit it by providing exclusive access to objects within a session. Infospheres supports service group multiplicity. Aliases are also supported but in the more traditional form as described by Bayerdorffer (see Section 3.3.1). The name binding model would ideally be of the category **bDmAsSr**, where both **s** and **S** are included, but shall be classified as **bDmASr** for simplicity with the additional note that although name sharing is possible it is not guaranteed.

Infospheres supports an exact matching between an object identifier (a URL) and the required object. Each object is required to register the location of a home URL and a service type. The usage of a home location model introduces cached information. The name resolution model for Infospheres is:

$$\begin{aligned}
\prec_R &: \textit{open} \prec_R \textit{closed} \\
\prec_M &: \textit{exact} \\
\prec_P &: \textit{unambiguous} \\
\prec_T &: \textit{cached} \prec_T \textit{authoritative}
\end{aligned}$$

Java Remote Method Invocation

Java RMI is a distributed object system designed to support small scale client/server communication using an ORB-style model. Java RMI's name binding model supports active rebinding, service group multiplicity and a restricted form of dynamic aliasing. Aliases can only be created by registering the server object multiple times. The name binding model for a Java RMI system is **BDmASr**.

The name resolution model for Java RMI is very simple and is used to map a unique name to an object. The name resolution model is:

$$\begin{aligned}
\prec_P &: \textit{unambiguous} \\
\prec_M &: \textit{exact}
\end{aligned}$$

A Java RMI system combined with LDAP extensions provides a form of attribute-based naming with generic support for the specification of optional and mandatory attributes. Cached information is also generated due to the propagation of LDAP information throughout the network. A Java RMI system using LDAP and attribute-based naming has the following name resolution model:

$$\begin{aligned} \prec_P &: \textit{ambiguous} \prec_P \textit{unambiguous} \\ \prec_Y &: \textit{optional} \prec_Y \textit{mandatory} \\ \prec_M &: \textit{partial} \prec_M \textit{exact} \\ \prec_T &: \textit{cached} \prec_T \textit{authoritative} \end{aligned}$$

Legion

Legion is a metacomputing system designed to support global scale communications. Legion supports one static alias per object and dynamic semantic description. The name binding model for Legion is **bDmASr**.

The name resolution scheme for Legion is attribute-based and uses cached data. Data that may be ambiguous or unambiguous. Legion provides a match-based ordering of attribute selection. The name resolution scheme for Legion has the following preferences:

$$\begin{aligned} \prec_P &: \textit{ambiguous} \prec_P \textit{unambiguous} \\ \prec_M &: \textit{partial} \prec_M \textit{exact} \prec_M \textit{unique} \\ \prec_T &: out-of-date \prec_T \textit{cached} \prec_T \textit{authoritative} \end{aligned}$$

4.4 Summary

The name binding and resolution models presented by Bayerdorffer and Bowman *et al* enable a classification of naming characteristics for mobile and distributed object systems by taking into account commonalities in binding and resolution requirements. These models lay the groundwork for extension as they define not only a model but a methodology for constructing future models.

The classification of existing systems shown in this chapter explores the suitability of the classifications for these types of systems. What is shown is that the classifications, in particular within the name binding model, are too broad and need to be narrowed to correctly classify systems while differentiating systems that have different models. The definitions of soundness and completeness can be used as an analog to this broadness. The naming models described in this chapter are complete, in that they represent a classification of all matching naming systems with the addition of systems that are incorrectly classified. The intention of this thesis is to develop naming models that provide a sound classification and represent only correct classification.

The attributes of mutability and multiplicity are specific examples of this problem. Mutability is seen in two forms: name reuse and active rebinding. These two categories have very different implications on the operation of the system and the use of naming. For example, active binding can be applied frequently and within the object's lifetime, while name reuse can only be applied after specified time periods to ensure that the initially bound object no longer exists.

Service group and replication multiplicity also have implications on the operation of a system. Service group multiplicity is found in systems where multiple objects have the same name and can equally perform the same task. Replication multiplicity is generally found in systems that allow object grouping or explicitly support replication.

Chapter 5 takes this discussion further and proposes extensions to Bayerdorffer's and Bowman *et al*'s models that refine the classification. These extensions also take into account elements that are missing from the classification.

Table 4 presents a summary of the support for attributes defined in Bayerdorffer's name binding model. As can be seen in this summary, certain attributes (specifically aliasing, dynamic domain and name sharing) are commonly supported while others (description) are infrequently supported. This is also seen in Table 5, which presents a summary of the resolution preference support for the existing systems examined in this chapter. A $\surd$ within these tables indicates that the system in question supports the specified attribute, while a $\times$ indicates that the system supports the orthogonal pair of the attribute. For example, a $\times$ for the multiplicity attribute indicates that the system supports static binding.

System	Mutability	Dynamic Domain	Multiplicity	Aliases	Shared Names	Description
Aglets	×	√	×	×	√	×
Ajents	√	√	×	√	√	×
Aleph	√	×	√	×	×	√
CORBA	×	√	×	√	√	√
d'Agents	×	√	×	√	√	×
DCE	×	√	×	√	√	×
DCOM	×	√	×	√	√	×
Emerald	√	√	×	√	√	×
Globe	×	√	√	√	√	×
Globus	√	√	×	√	√	√
Grapevine	√	√	√	√	√	×
Infospheres	×	√	×	√	√	×
Java RMI	√	√	×	√	√	×
Legion	×	√	×	√	√	×
MOA	√	√	×	√	√	×
Mobile Process	√	√	×	×	√	×
Mobile IP	√	×	×	√	√	×
MOLE	×	√	×	√	√	√
Obliq	×	√	×	√	√	×
Regional Directories	√	×	×	√	√	×
Sumatra	×	√	×	√	√	×
Telescript	×	√	√	×	√	×

Table 4: Existing object system support for the name binding model.

System	Registered	Mutability	Precision	Yellow Pages	Match Based	Voting	Temporal
Aglets	√	×	√	×	√	×	√
Ajents	×	×	√	×	√	×	×
Aleph	×	×	×	√	√	×	√
CORBA	√	×	√	√	√	×	√
d'Agents	×	×	×	√	√	×	√
DCE	√	×	√	√	√	×	√
DCOM	√	×	√	×	√	×	√
Emerald	×	×	√	×	√	×	√
Globe	×	×	√	×	×	×	√
Globus	√	√	√	√	√	×	√
Grapevine	√	×	√	×	√	×	√
Infospheres	√	×	√	×	√	×	√
Java RMI	×	×	√	×	√	×	×
(w LDAP)	×	×	√	√	√	×	√
Legion	×	×	√	×	√	×	√
MOA	×	×	√	×	√	×	√
Mobile Process	√	×	√	×	√	×	√
MOLE	×	×	×	√	√	×	√
Obliq	×	×	√	×	√	×	√
Regional Directories	×	×	√	×	√	×	√
Sumatra	×	×	√	×	√	×	√
Telescript	√	×	×	×	√	×	√

Table 5: Existing object system support for the name resolution model.

Chapter 5

The Extended Naming Model

The name binding and resolution models described in Chapter 3 represent a general classification mechanism for object systems. The classifications of existing mobile and distributed object systems presented in Chapter 4 have emphasised that the name binding model is too broad to provide a complete classification of current mobile and distributed object systems and does not provide enough differentiation between systems which, in practice, have very different semantics and behaviour. Additionally, both models are lacking in their classification of transparency and the distributed nature of many current naming systems. This chapter presents extensions to these models that will allow full classification of existing object systems with respect to their support for flexible naming and transparency. The semantic effect of each attribute introduced within the classification is discussed at length. A formal definition of each extension is provided.

An examination of the extended classification model introduces three categories within a naming model: name binding, name management and name resolution. Each attribute and preference defined in the extended classification model is reclassified according to which category it belongs.

The extended naming model is then used to reclassify the examples of mobile and distributed object systems presented in Chapter 4. The DISCWorld ORB system's naming model is then introduced with a formal classification according to the extended models. A discussion of the structure and semantics of names within the DISCWorld ORB system is presented along with the impact of multi-level distributed directory systems and scoping on the naming requirements.

5.1 Extensions to the Existing Models

As outlined in Chapter 4, there are obvious extensions and refinements to the classification models for name binding and resolution. Due to the broad nature exhibited by some attributes the level of detail in existing classifications is too shallow. By extending and refining these offending attributes the classification depth can be extended to classify current object systems in sufficient detail.

Chapter 2 outlines the location and relocation support present in existing mobile and distributed object systems. Central to this discussion is the support for location dependence as well as location and relocation transparency, of which no mention is made in the existing classification models. Additionally, the existing classifications are not able to describe multiple-level naming systems. Existing distributed and middleware systems support multiple levels of naming and scoping, including tree-based and peer-based systems. In a hierarchical system, resolution can also be limited to certain levels of hierarchy or distances within the hierarchy. The ability to describe these characteristics is also missing from the existing classifications.

5.1.1 Extended Name Binding Model

Name binding, as defined by Bayerdorffer [15, 16], encompasses the rules governing how names can be entered into a system and how these names can be altered during the lifetime of the object and, also, the system. The extensions to the name binding model presented in this chapter can be divided into extensions to, and refinement of, existing attributes and the introduction of new attributes.

Refinement of Existing Attributes

As outlined in Chapter 4, the attribute of mutability is defined in two forms: active rebinding and name reuse. Active rebinding, name reuse and static binding are orthogonal. Active rebinding can be defined as the mutability of a name that is bound to an object still resident and active within the system, whereas name reuse requires that the initially bound object be inactive and no longer known by the system.

In many systems, active rebinding requires that all current connections to the object be seamlessly rebound to the new object [6, 85, 95, 161]. This requirement is usually implemented using one of two mechanisms. The first mechanism is a requirement that no connections can be present before an active rebinding. This restriction is similar to that found in migration in many mobile object systems [33, 34, 95]. In these systems, migrating objects are unable to begin migration until they reach the state of having no connected references and no queued invocations. The second mechanism is the use of an intermediate object that manages all incoming connections and can perform any required protocol translations. Examples of an appropriate intermediate object include a home location or a skeleton[8]. Java RMI [55, 167] is an example of a system that does not provide strict active rebinding; this system allows existing references to be used after a rebinding.

[8]The alternative forms of a skeleton, such as a scion or an object adapter, are equivalent in this ability.

Name reuse requires the ability to identify when the initially bound object is no longer in existence. Some systems overcome this through a general reuse time period, others overcome the problem by tracking each object within the system. Some systems, such as CORBA [18, 136, 138] and MOA [131], allow each object to be explicitly unbound before being removed from the system. Explicit unbinding allows clients to rebind a name regardless of the reuse time period.

A redefinition of the mutability attribute is required that encompasses the three orthogonal forms of mutability. The extended definition of the mutability attribute is:

- Mutability of bindings: a name that may be rebound to another object during the initial object's life-time can be *actively rebound* ($\mathbf{B^a}$). A name that can be rebound to another object during the lifetime of the system, but not during the initial object's life-time, can be *reused* ($\mathbf{B^r}$). A name that can not be rebound is *statically bound* ($\mathbf{b}$).

Multiplicity is also present in existing mobile and distributed object systems in two forms: service group multiplicity and replication multiplicity. Service group multiplicity requires that each object be identified by a service type. The multiplicity found in this form allows a name to refer to all objects matching a specific service type, but only one is selected upon resolution. Replication multiplicity allows one name to refer to multiple objects with all objects involved in a resolution. In a system that supports replication multiplicity, this may mean that all objects in the one group are migrated at the same time [1, 151] or that all objects receive the same communications [8, 10, 176–179].

The extended multiplicity attribute is not strictly orthogonal. As has been seen in the examination of the Sumatra [1, 151] and MOLE [14] systems, it is possible to exhibit both the attributes of replication and service group multiplicity. These attributes are, however, orthogonal to singularity. These extended attributes, along with the state of singularity, can be defined as follows:

- Multiply-bound names: if a name may be bound to multiple objects and each of those objects is part of each resolution then the naming model has the property of *replication multiplicity* ($\mathbf{M^r}$). If a name may be bound to multiple objects and one of those bound objects may be resolved then the naming model has the property of *service group multiplicity* ($\mathbf{M^s}$). If a name may be bound to a single object only then the naming model has the property of *singularity* ($\mathbf{m}$).

The attribute of aliasing is also present in two forms: static aliasing and dynamic aliasing. These two forms have distinct influences on the semantics of the system they are present in. For example, a static aliasing system allows an object to have a set of aliases defined for it at its inclusion within a system. This enables the object to form part of multiple service groups and to have multiple access points. A dynamic aliasing system also enables objects to have multiple service groups and access points, and extends this support by the ability to change this information according to the current state of the object and its environment.

An example of the usage of dynamic aliasing is a system where the set of aliases defines the set of single-client entry points to an object and hence defines the number of clients that the object can service at one time. If, during the object's lifetime, its environment changes or it is overloaded by client requests, the object is able to add or remove entry points to satisfy its new requirements. Examples of systems with similar behaviour include Infospheres [33, 34] and Grapevine [161]. A redefinition of the alias attribute to encompass the three orthogonal states of static aliasing, dynamic aliasing and no aliasing is:

- Multiply-named objects: if an object may have multiple statically defined names then the name system supports *statically aliased names* ($\mathbf{A^s}$). If an object may have multiple dynamically defined names then the name system supports *dynamically aliased names* ($\mathbf{A^d}$). If an object may have only one name then the name system supports *unaliased names* ($\mathbf{a}$).

Descriptive naming is found in the forms of state-based description and semantic description. State-based description allows the inclusion of dynamic state information into the naming system. This form of naming can be used to specify such information as the current state of access, the load on the object or its current platform requirements. Semantic description is used to define what the object does. This form of description can be used to annotate a name with information regarding the object's service type, its class hierarchy, and method signatures. The new definition of descriptive naming that encompasses the three orthogonal states of state-based description, semantic description and non-description is:

- Descriptive names: names that can carry information about the state of objects to which they are bound are termed *state-based descriptive names* ($\mathbf{R^b}$). Names that can carry information about the semantics of the objects to which they are bound are termed *semantic descriptive names* ($\mathbf{R^s}$). Names that can not carry additional information are termed *non-descriptive names* ($\mathbf{r}$).

An implied extension to this definition is an attribute that defines the static nature of defined attributes. For example Globus [42, 54, 57, 58] supports static semantic description, while CORBA supports dynamic semantic description, and Aleph [43, 84–86] supports dynamic state-based description. A definition that encompasses both static and dynamic attribute specification is:

- Dynamic attributes: for naming models that support attribute resolution, if the attribute may be changed dynamically, the naming model supports *dynamic attributes* ($\mathbf{F}$), otherwise, if attributes are statically defined upon object creation the naming model supports *static attributes* ($\mathbf{f}$).

The attribute of name sharing defines whether multiple clients can concurrently use the same name. Name sharing implies that it is possible for named objects to be concurrently accessed, through the same name, by multiple clients. In a system that supports private names, each bound name can only be used by a single client. In addition to these specifications, a common example of name sharing found in existing mobile and distributed object systems is when a shared name is used as an access point to multiple, potentially private, entries. In these systems an entry acts as a direct communication point, while a name acts as an abstraction. In some models, an entry will be equivalent to a name and hence the attribute of name sharing will correspond to this attribute. An entry may be assigned to a specific client or entries may be shared by multiple clients.

A naming system that supports shared names may, at the entry level, require that each entry support only one client. A restricted set of entries then sets a maximum limit on the number of clients that can be accepted. This form of model is commonly found in fragmented object or ORB models where skeletons are used to gain access to entry points within the object. As an example, Infospheres supports multiple private inboxes as entry points that are accessed through a shared name that references a home location.

The definition of the entry sharing attribute defines an orthogonality between shared entries and private entries. The definition is:

- Entry sharing: an object that requires a unique entry point for each client supports *private entry* (**E**), otherwise if entry points may be shared the object supports *shared entry* (**e**).

Additional Attributes

In addition to the attributes defined by Bayerdorffer, there are several characteristics of names within mobile and distributed object systems that are particularly important. These characteristics define how closely linked each name is to its environment and the objects which it is currently communicating with. By defining new attributes to represent these characteristics it is possible to extend the naming model to fully classify the naming system of such an object system.

The uniqueness of a name is characterised by its ability to have multiple objects bound to it through multiplicity. The structure of a name also has an effect on its uniqueness. For example names that are rigidly structured are often contextual names. Contextual names consist of the combination of a path of names that represent the hierarchical path to the referenced object. For example the name *edu.adelaide.cs.katrina* might represent the object named *katrina* within the domain *cs*, which is itself within the domain *adelaide* and so on.

A contextually unique name will not necessarily be unique on a global scale, unless its complete hierarchical name is used (as in Globus). A globally unique name is, by definition, unique in any domain. The consequence of a naming system that requires globally unique names is that the implementation of the naming system must be able to ensure that each name is unique. This becomes a difficult problem in a distributed object system where multiple naming system objects are capable of providing names.

An attribute that defines the contextual basis of names can be used to infer the distribution properties and requirements of a naming system. For example, when combined with multiplicity, global naming allows any name to be created anywhere; when combined with singularity, global naming requires detailed support to ensure uniqueness. A definition that encompasses the orthogonal states of contextual naming or global naming is:

- Contextual naming: if a name is contextually structured and is part of a larger hierarchically structured name then that naming system supports *contextual naming* (**C**). If the name is not contextual, but exists within a global namespace then the naming system supports *global naming* (**c**).

Location independence defines the level of abstraction within a name with regard to its location. Systems with location independent naming generally have greater support for mobility as client references are not dependent on the current location of the mobile object. Combined with mutability, location dependent names can still be accessed successfully in a mobile system (see Section 2.1). This form of naming requires all issued names to be updated upon object migration.

A system that supports location independence is not required to update each name instance within the system each time the object migrates. An implication of this is that such a system has to provide mechanisms to resolve a name to an object regardless of the object's current location. A definition of the attribute of location independence is:

- Location independence: if a name does not require or contain knowledge about the location of the referenced object, then it is *location independent* (**I**), otherwise it is *location dependent*, (**i**).

Location transparency is a similar attribute to location independence. The difference is that although a location transparent name can contain location information, it can not present that information to the user of the name. This also means that specifying the location of an object can not be a necessary part of name resolution. Location transparency is a common attribute of systems where the scale of distribution or mobility is large [29,102] or relocation transparency is also a requirement [75,176]. The definition of an attribute of location transparency, orthogonal to location opaqueness, is:

- Location transparency: if a name does not expose the location of its referenced object it is *location transparent* (**T**), otherwise it is *location opaque* (**t**).

The attribute of relocation transparency is present in systems where a name is valid regardless of the object's location. At any point during communication between a client and the object the name must be valid. The implication of this upon a mobile object system is that each name must be valid at any point during connection to the referenced object, during the referenced object's migration, and after the referenced object has successfully taken up residence at its new location. As described in Chapters 1 and 2, this problem is a difficult problem to solve for large scale systems.

An attribute that defines the orthogonality between relocation transparency and relocation opaqueness can be defined as:

- Relocation transparency: if a name maintains its validity regardless of the location of the referenced object it is *relocation transparent* (**O**), otherwise it is *relocation opaque* (**o**).

5.1.2 Extended Name Resolution Model

Bowman *et al* [25, 26] define several approximation functions and preferences by which the name resolution process can be ordered and modelled. These functions define how the client wishes its results to be ordered and also how results are ordered within the database. The classification model defined by Bowman *et al* can be viewed as a framework for defining preferences that suit the system that is being modelled. This framework is designed to be extended through the creation of new preference orderings and approximation functions. Resolution functions, able to apply any subset of approximation functions, can then be applied to a resolution to produce an appropriately selected and ordered set of results.

The preferences defined by Bowman *et al* in [26] present a valid classification of most aspects of name resolution systems for mobile and distributed object systems. All attributes except for one are commonly found in the naming systems of existing mobile and distributed object systems; the exception being the voting preference. The preferences defined describe client and database preferences for temporally separated data and also data that is an optional or mandatory part of the database.

The discussion of support for the mutability preference in Chapter 4 motivates the inclusion of a variation of the mutability preference with an opposite ordering. This variation is defined here as the currency preference, where a preference is defined for attributes that may change and hence may represent the current state of an object, over those that are statically defined or can only be defined once. The currency preference, $\prec_C$, is defined as:

$$\textit{static} \prec_C \textit{dynamic}$$

Preferences specific to a distributed name resolution system are not defined by Bowman *et al.* A distributed name resolution system may have a preference based on where the object is within the distributed system, hence placing a preference on objects that can be resolved within a subset of the distributed nodes.

To extend the name resolution system as defined by Bowman *et al*, two new preferences are defined based on different forms of locality. These preferences exist as preferences for use in a database approximation function. The first new preference places a preference for objects that are local rather than remote. This preference is found in distributed naming systems where the distribution is peer-based, (*i.e.* not hierarchical or tree structured). Examples of such systems include Grapevine [161], CORBA and Obliq [29, 30, 94]. This preference is defined as the locality preference.

Locality preference. The locality preference $\prec_L$ defines a preference for attribute or object information defined locally over attribute or object information defined remotely, where:

$$remote \prec_L local$$

The second new preference places a preference on the locality within a tree-based or hierarchical directory system. This system places a preference for objects on the local node over those objects within one node distance, and then over expanding node distances. This preference is applicable to systems with a distributed naming or resolution system, such as d'Agents [71,72,110] yellow pages server or MOA's hierarchical name server. This preference is defined as the node preference.

Node preference. The node preference $\prec_N$ defines a preference for objects on the same node over those at remote nodes. The node preference can be defined by:

$$multi{-}hop \prec_N one{-}hop \prec_N same$$

Within a hierarchical system each node within the system may be organised as part of a tree and may be either a parent node or a leaf node. The node distance is the distance, in terms of uni-directional hops, within the tree that communications have to travel. In the case of resolution, node distance implies a distance that the resolution process has to proceed throughout the tree before the name can be resolved. The node preference is defined as a generic preference model to be applied to a system. For example, a node preference for a system with three node levels may have a preference of:

$$\prec_N :\ level3 \prec_N level2 \prec_N level1$$

An example of system with a node preference is d'Agents. d'Agents has a three level name resolution system where resolution attempts proceed from the local node to specific nodes within the system and then to network nodes. A node preference for d'Agents is:

$$\prec_N :\ network \prec_N remote{-}node \prec_N local$$

A temporal preference is similar in concept to these locality-based preferences. A temporal preference places a preference on data that is authoritatively known over information that is second or third hand knowledge. In practice, this often translates to information that is directly known by the node because the object in question is resident on the node. The difference between a locality-based preference and a temporal preference is that they place a direct preference on the relative location of the object rather then the location of the information describing the object.

5.1.3 Formal Definitions

The extended name binding and resolution models can be represented by formal definitions defining the binding functions required to implement the new attributes. The formal definitions for these attributes and their binding functions are similar to those defined in Section 3.3.3 and represent functions that can be applied upon each binding to validate the binding with regard to a naming model. These functions are functions on the domain of names **N** and objects **O** known within the system, and perform operations on the database **D** that contains all of the bindings known to the system.

For the extended mutability attribute there are now two cases of mutability to be defined. With a system supporting active rebinding an alteration in the binding between a name and its object can be performed if the binding exists. A system that supports name reuse can only alter a binding if the object to which the name is currently bound no longer exists within the set of objects that are currently known within the system. A system that supports static binding can not change a binding that already exists within the database. These requirements can be formally defined by the following two definitions and the given definition for static binding (see Definition 3.2).

Definition 5.1 (Active Rebinding Function). A name binding function supporting active rebinding of names for objects, is defined over the domain of names **N** and the domain of objects **O** as a function $f_{B^a} : N \times O \rightarrow N \times O$, where for a database of bindings **D**, and a name n_i and an object o_i to be bound:

1. if $\exists n_j \in \mathbf{N}$ where $n_i = n_j$, then:
   (a) $f_{B^a}(n_i, o_i) = \mathbf{D} \setminus (n_j, o_j)$, and
   (b) $f_{B^a}(n_i, o_i) = \mathbf{D} \times (n_i, o_i)$
2. else $f_{B^a}(n_i, o_i) = \mathbf{D} \times (n_i, o_i)$.

In the name reuse binding function, it is the responsibility of the naming system to ensure that the set of objects **O** is maintained with respect to those objects that are deemed to be known or active within the system. This separation of concerns is due to the varying definitions of an active object and the varying rules governing the transition between activity and inactivity within mobile and distributed object systems.

Definition 5.2 (Name Reuse Binding Function).
A name binding function supporting name reuse for names and objects, is defined over the domain of names **N** and the domain of objects **O** as a function $f_{B^r} : N \times O \to N \times O$, where for a database of bindings **D**, and a name n_i and an object o_i to be bound:

1. if $\exists n_j \in \mathbf{N}$ then
   (a) if $\exists o_j \in \mathbf{O}$ then $\to failure$
   (b) else $f_{B^r}(n_i, o_i) = \mathbf{D} \times (n_i, o_i)$
2. else $f_{B^r}(n_i, o_i) = \mathbf{D} \times (n_i, o_i)$.

The main difference between these two definitions is that the active rebinding function allows rebinding of names for all bound objects; the status of the currently bound object is unimportant. The name reuse binding function only allows rebinding if the currently bound object is no longer known within the object system.

The multiplicity attribute also introduces two new definitions that represent replication multiplicity and service group multiplicity. For replication purposes, each object sharing the name must be of the same *type* or class; for service group multiplicity, each object sharing the name must provide the same service. An integral part of these definitions are two functions that can determine certain attributes of an object. A function, *type*, is introduced in order to support replication multiplicity. This function is required to return *true* if two objects are of the same type. The details of the *type* function are dependent on the object system and are separated from the implementation of the naming system. This separation enables the development of *type* functions that are able to determine equality regardless of the implementation language or version of the objects to be compared. A function to determine the service properties of an object is also required that returns *true* if two objects provide the same service. This function, *service*, can be as simple as a function that performs a comparison of the service type attributes linked to an object. The definitions of the binding functions for the multiplicity attribute are as follows:

Definition 5.3 (Replication Multiplicity Binding Function). A binding function for replication multiplicity over the domain of names **N** and the domain of objects **O** is a function $f_{M^r} : N \times O \to N \times O$ where for a database of bindings **D**, and a name n_i, an object o_j to be bound, and an object o_i already bound to n_i:

1. if $type(o_i) = type(o_j)$ then $f_{M^r}(n_i, o_j) = \mathbf{D} \times (n_i, o_j)$
2. else $\to failure$.

Definition 5.4 (Service Group Multiplicity Binding Function). A service group multiplicity binding function over the domain of names $\mathbf{N}$ and the domain of objects $\mathbf{O}$ is a function $f_{M^s} : N \times O \rightarrow N \times O$ where for a database of bindings $\mathbf{D}$, a name n_i, an object o_j to be bound, and an object o_i already bound to n_i:

1. if $service(o_i) = service(o_j)$ then $f_{M^r}(n_i, o_j) = \mathbf{D} \times (n_i, o_j)$
2. else $\rightarrow failure$.

The two forms, static aliasing and dynamic aliasing, can also be formally defined as binding functions. Static aliasing requires that the object to be bound be new to the system, hence it can not already be contained within the set of objects $\mathbf{O}$. This requirement implies that all bindings of the object to its aliases must be an atomic operation; no changes are made to the database until all bindings are complete. A dynamic aliasing function allows aliasing at any point during the object's lifetime.

Definition 5.5 (Static Alias Binding Function). A static alias binding function over the domain of names $\mathbf{N}$ and the domain of objects $\mathbf{O}$ is a function $f_{A^s} : N \times O \rightarrow N \times O$ where for a database of bindings $\mathbf{D}$, and a name n and an object o to be bound:

1. if $o \notin \mathbf{O}$ then $f_{A^s}(n, o) = \mathbf{D} \times (n, o)$
2. else $\rightarrow failure$.

Definition 5.6 (Dynamic Alias Binding Function). A dynamic alias binding function over the domain of names $\mathbf{N}$ and the domain of objects $\mathbf{O}$ is a function $f_{A^d} : N \times O \rightarrow N \times O$ where for a database of bindings $\mathbf{D}$, and a name n and an object o to be bound, $f_{A^d}(n, o) = \mathbf{D} \times (n, o)$.

The descriptive names attribute also introduces two new definitions, one for a state-based description and one for a semantic description. The only change required in the formal definition of a descriptive names binding function is a change from the addition of state information to the addition of semantic information. State and semantic information can be represented in the same form, hence, the only required change in the formal definition of the binding function is to replace the state information with a general information form, attribute information for example. This binding function can then be used for both state-based and semantic descriptive name binding.

Definition 5.7 (Descriptive Name Binding Function). A descriptive name binding function over the domain of attributes $\mathbf{A}$, the domain of names $\mathbf{N}$ and the domain of objects $\mathbf{O}$ is a function $f_R : 2^A \times N \times O \rightarrow N \times O$ where for a database of bindings $\mathbf{D}$, a set of attributes $\mathbf{a}$, a name n and an object o to bind, $f_R(\mathbf{a}, n, o) = \mathbf{D} \times (\mathbf{a}, n, o)$.

The formal definition of the dynamic attribute specification is similar to that of static and dynamic aliasing. A system that supports static attribute specification can only alter the binding of (a_i, n_i, o_i) upon the object's entry into the set of known objects. A dynamic attribute system can alter attribute bindings at any stage during the object's lifetime.

Definition 5.8 (Dynamic Attribute Binding Function). A binding function for dynamic attribute binding over the domain of attributes **A**, the domain of names **N** and the domain of objects **O** is a function $f_F : 2^A \times N \times O \rightarrow N \times O$ where for a database of bindings **D**, and a set of attributes **a**, a name n and an object o to be bound, $f_F(\mathbf{a}, n, o) = \mathbf{D} \times (\mathbf{a}, n, o)$.

Definition 5.9 (Static Attribute Binding Function). A binding function for static attribute binding over the domain of attributes **A**, the domain of names **N** and the domain of objects **O** is a function $f_f : 2^A \times N \times O \rightarrow N \times O$ where for a database of bindings **D**, and a set of attributes **a**, a name n and an object o to be bound:

1. if $o \notin \mathbf{O}$ then $f_f(\mathbf{a}, n, o) = \mathbf{D} \times (\mathbf{a}, n, o)$
2. else $\rightarrow$ *failure*.

The contextual naming attribute implies semantic behaviour within a distributed naming system, rather than implication on the binding system. For example, a contextual naming system with the singularity attribute has only to ensure uniqueness for the local node of the naming system, whereas a global naming system has to ensure uniqueness over all nodes within the system. This attribute has no binding function defined; it requires definition through the design of the naming system itself as it is the combination of this attribute with other defined attributes that has effect.

The definitions of the attributes of location independence and location transparency require the definition of functions that determine whether location components are present in a name and whether they are visible. A function, *locate*, is required that returns *true* if a name contains a location component. Additionally a function, *visible_locate*, is required that returns *true* if the name's location element is transparent. Location dependent information can be stored in the contents of a name or reference. For example, a name may be resolved to a reference that acts as the active component of the name. This reference stores information such as protocol information and location information, and is commonly found as a stub or scion. A function that determines whether a reference is location independent must be able to determine whether the contents of the reference are also location independent.

The definitions of the *locate* and *visible_locate* functions are separated from the definition of the binding function as they are dependent on the type of location information used within the system. For example, a location can be specified by an IP address, a processor ID or a textual name (which translates to an IP address). For a particular object system, location functions can be determined that are able to identify the specific kind of location information used for that object system.

Definition 5.10 (Location Independent Binding Function). A binding function for location independent binding over a domain of names $\mathbf{N}$ and a domain of objects $\mathbf{O}$ is a function $f_I : N \times O \rightarrow N \times O$ where for a database of bindings $\mathbf{D}$, and a name n and an object o to be bound:

1. if $locate(n)$ then $\rightarrow failure$
2. else $f_I(n, o) = \mathbf{D} \times (n, o)$

Definition 5.11 (Location Dependent Binding Function). A binding function for location dependence over a domain of names $\mathbf{N}$ and a domain of objects $\mathbf{O}$ is a function $f_i : N \times O \rightarrow N \times O$ where for a database of bindings $\mathbf{D}$, and a name n and an object o to be bound, $f_i(n, o) = \mathbf{D} \times (n, o)$.

Location transparency can also be determined by examining a name for location information. In the case of location transparency, only the information openly presented by the name to the client need be examined.

Definition 5.12 (Location Transparency Binding Function). A binding function for location transparent binding over a domain of names $\mathbf{N}$ and a domain of objects $\mathbf{O}$ is a function $f_T : N \times O \rightarrow N \times O$ where for a database of bindings $\mathbf{D}$, and a name n and an object o to be bound:

1. if $visible\_locate(n)$ then $\rightarrow failure$
2. else $f_T(n, o) = \mathbf{D} \times (n, o)$

Definition 5.13 (Location Opaque Binding Function). A location opaque binding function over the domain of names $\mathbf{N}$ and the domain of objects $\mathbf{O}$ is a function $f_t : N \times O \rightarrow N \times O$ where $f_t(n, o) = \mathbf{D} \times (n, o)$.

The attribute of relocation transparency defines a semantic property of a system. The ability for references to relocate transparently is not necessarily connected to the reference's support for location independence or transparency and can not be determined by information contained or not contained within the reference. Accordingly there are no binding functions defined for relocation transparency and relocation opaqueness. A system with the property of relocation transparency has to provide mechanisms to support this transparency which may not depend on the structure or management of names. This is evidenced by the many, varied forms of relocation transparency as shown in Table 3 on page 48.

5.2 Model Categories

The attributes and preferences defined in Sections 3.3 and 5.1 are grouped into the categories of binding and resolution. The formal definition of the binding attributes has shown that some have effects on both name binding and resolution while others have an effect on the management and maintenance of names rather than on binding.

The name model of a system can then be defined as three categories: name binding, name management and name resolution. Name binding consists of those attributes defined by Bayerdorffer and extended here with the exception of the mutability-based attributes. The mutability-based attributes and their formal binding functions are within the name management category. Name resolution is itself divided into two subcategories: resolution preferences and resolution attributes. The resolution preferences consist of those defined for the client and approximation functions. The resolution attributes are separately defined as they have effects on the name binding database **D**.

Mutability is an example of an attribute that has no real effect on name binding. A name can be rebound to another object depending on the current state of its initial binding: hence this attribute is more appropriately placed into the category of name management. Relocation transparency is another attribute that has more influence on name management than on name binding or resolution.

To successfully apply name management attributes upon each alteration to a $(name, object)$ binding, a management function must be defined that applies those attributes defined for the name management model. Given that the name management model currently contains only one attribute with explicit effect on the database, this function is defined more for compatibility and extension purposes than for any specific requirement in the current model. A complete management function is similar to a complete binding function in that it takes the name management model defined by a set of attributes and upon each management application, the function applies its set of attribute functions to verify that either the management succeeds or fails. A complete management function can be defined as:

Definition 5.14 (Complete Management Function).
A complete Management function can be defined as an induced preference order $\prec$ on a set of name management functions $\Pi_m = \{\pi_1, \pi_2, ..., \pi_n\}$ as $\rho_m : 2^N \times 2^O \to 2^N \times 2^O$ over a domain of names **N**, a domain of objects **O** and operates on the database of name bindings **D**, by $\rho_m(\mathbf{n}, \mathbf{o}) = \pi_i(\mathbf{N}, \mathbf{O})$, where for $i \in 1..n$:

1. $\pi_i(\mathbf{N}, \mathbf{D}) \neq failure$, if $i < n$, and

2. $\forall \pi_j \in \Pi_m[\pi_i \prec \pi_j$ implies that $\pi_n(\mathbf{N}, \mathbf{D}) = \perp]$.

The definition of binding functions for the shared names and descriptive names attributes also has effect on name resolution. Definitions 3.10 and 3.12 (page 70) define resolution functions for shared names and private names that are more appropriately applied when combined with name resolution preferences. A resolution approximation function that takes the preferences exhibited within a model can apply these preferences and then apply the necessary resolution functions to perform the resulting operations on the name binding database. For example, a resolution model that requires private naming will invoke a resolution function on the set of objects returned from applying the resolution preferences to the database. In this form of mutually exclusive access system, the private name resolution function will ensure that the object is accessible (*i.e.* it is not already in use) and perform any reorganisation of the database to change the name's state to "used".

The attribute of descriptive names introduces resolution functions, which are defined as Definitions 3.14 and 3.16 (page 71). These resolution functions can be applied along with the resolution preferences to select objects with appropriate attributes as defined by the client.

The introduction of resolution functions into the resolution stage of a name model requires that a function be introduced that is capable of applying the resolution functions. Similar to the complete binding function, a complete resolution function must apply the preferences given by the resolution model and then apply the resolution functions to perform any required tasks on the database. A complete name resolution function that applies all name resolution attribute functions has no affect on the database. A complete resolution function can be defined as:

Definition 5.15 (Complete Resolution Function). A complete resolution function given a set of attributes (or names) $\mathbf{A}$, and a set of selected objects $\mathbf{O}$, that operates on a database of name and object bindings, $\mathbf{D}$ is a function $\rho_r : 2^A \times 2^O \rightarrow 2^O$. If given a set of resolution functions $\Pi = \{\pi_1, \pi_2, ..., \pi_n\}$ that define the model of the complete resolution function then for $i \in 1..n$:

1. $\pi_i(\mathbf{N}, \mathbf{D}) \neq failure$, if $i < n$, and
2. $\forall \pi_j \in \Pi[\pi_i \prec \pi_j$ implies that $\pi_n(\mathbf{N}, \mathbf{D}) = \mathbf{D}]$.

5.3 Reclassification of Existing Systems

A reclassification of the systems examined in Chapters 2 and 4 can be performed with respect to the extended name models. A reclassification of a system according to the extended models defines four classification sets: the name binding model, the name management model, the name resolution preference model and the name resolution attribute model. To focus on the changes in classification, only those systems that exhibit some of the new classifications are presented.

5.3.1 Mobile Process Systems

Mobile process systems exist in the initial binding category of **BDmaSr**. The form of mutability is name reuse. In addition to these features, mobile process systems support location and relocation transparency. Names are generally globally unique, although there are some exceptions. The extended name binding model is **DmaSeciT**. The name management model is $\mathbf{B^{r}O}$. The name resolution model is unchanged except for the addition of the attribute set **r**.

5.3.2 Mobile Object Systems

Ajents

The initial name binding model for Ajents [95] is **BDmASr**. Ajents supports dynamic aliasing and active rebinding but does not support any form of independence or transparency. The name binding model using the extended classification is $\mathbf{DmA^{d}Secit}$. The name management model is $\mathbf{B^{a}o}$.

The name resolution model for Ajents is unchanged except for the addition of the attribute set **r**.

d'Agents

The name binding model for d'Agents is **bDmASr**; the extended name binding model is $\mathbf{DmA^{d}SeCit}$. The d'Agents system supports dynamic aliasing and also contextual naming. Each name consists of a reference to the object and a name defining the location of the object. The name management model is **bo**.

The d'Agents system uses three stages of distribution in its resolution mechanism. Resolution is first attempted on the local node, and then on a selection of remote nodes within the system. If these resolution attempts are unsuccessful then resolution is attempted off the network. This behaviour introduces the node preference. The extended name resolution model for d'Agents is:

$$\begin{aligned}
\prec_Y &: \ optional \prec_Y mandatory \\
\prec_M &: \ partial \prec_M exact \\
\prec_T &: \ out{-}of{-}date \prec_T cached \prec_T authoritative \\
\prec_N &: \ network \prec_N remote{-}node \prec_N local
\end{aligned}$$

with the addition of the attribute set **r**.

Emerald

The existing name binding model for Emerald [21,91,102] is **BDmASr**. Emerald supports location independent and transparent referencing and relocation transparency. Names are globally defined. The extended name binding model for Emerald is $\mathbf{DmA^{d}SecIT}$. The name management model is $\mathbf{B^{r}O}$. The name resolution model is unchanged except for the addition of the attribute set **r**.

MOA

MOA has an initial name binding model of **BDmASr** supporting active rebinding and static aliasing. Names are relocation transparent, but interestingly are neither location independent or transparent. Names are globally unique, giving an extended name binding model of $\mathbf{DmA^{s}Secit}$. The name management model is $\mathbf{B^{a}O}$.

Name resolution within a MOA system is performed in three stages. The initial stage is through direct, local resolution. The second stage is through use of the hierarchically structured name server system. The third stage is by contacting the object's home location. This introduces the node preference. The extended name resolution model for MOA is:

$$\begin{array}{rcl} \prec_M & : & exact \\ \prec_P & : & ambiguous \\ \prec_T & : & out\!-\!of\!-\!date \prec_T cached \prec_T authoritative \\ \prec_N & : & home\!-\!location \prec_N name\!-\!server \prec_N local \end{array}$$

with the addition of the attribute set **r**.

Obliq

The initial name binding model for Obliq is **bDmASr** with dynamic aliasing. Names are globally unique, location independent and transparent. Relocation transparency is also supported. The extended naming model is $\mathbf{DmA^{d}SecIT}$. The name management model is **bO**.

Obliq supports a distributed environment with a centralised name server. This name server provides a locality preference for locally resolvable names over remotely resolvable names. The extended name resolution model for Obliq, with the attribute set **r**, is:

$$\begin{array}{rcl} \prec_M & : & exact \\ \prec_P & : & unambiguous \\ \prec_T & : & out\!-\!of\!-\!date \prec_T cached \prec_T authoritative \\ \prec_L & : & remote \prec_L local \end{array}$$

5.3.3 Mobile Host Systems

Regional Directories

The initial name binding model for Regional Directories [5, 6] is **BdmASr** with support for active rebinding. Names are contextually structured and provide no form of location transparency or independence, but relocation transparency is provided. The extended model is $\mathbf{dmA^dSeCit}$. The name management model is $\mathbf{B^aO}$.

The hierarchically structured directory system supported by Regional Directories exhibits an example of a node-based preference. The extended name resolution model is:

$$\begin{aligned}
\prec_M &: \textit{exact} \\
\prec_P &: \textit{unambiguous} \\
\prec_T &: \textit{cached} \prec_T \textit{authoritative} \\
\prec_N &: \textit{higher}-\textit{level} \prec_N \textit{next}-\textit{level} \prec_N \textit{same}-\textit{level}
\end{aligned}$$

with the addition of the attribute set **r**.

5.3.4 Distributed Object Systems

Aleph

Aleph supports an initial binding model of **BdMasR**. With the addition of globally unique names and location and relocation transparency the extended name binding model is **dMasEciT**. The name management model is $\mathbf{B^aO}$. The name resolution model is unchanged except for the addition of the attribute set $\mathbf{R^bF}$.

CORBA

The initial name binding model for CORBA is **bDmASr** with dynamic aliasing. Each CORBA object has a location transparent reference. The adapter(s) provided for each registered CORBA object allow definition of one adapter entry per client. The extended name binding model is $\mathbf{DmA^dSEciT}$. The name management model is **bo**.

With the addition of a trader, CORBA systems exhibit a locality preference. Names can be resolved by a local ORB, and if this fails, further resolution can be attempted through a trader system. The extended name resolution model is:

$$\begin{aligned}
\prec_R &: open \prec_R closed \\
\prec_Y &: optional \prec_Y mandatory \\
\prec_P &: ambiguous \prec_P unambiguous \\
\prec_M &: partial \prec_M exact \\
\prec_T &: out{-}of{-}date \prec_T authoritative \\
\prec_L &: remote \prec_L local
\end{aligned}$$

with the addition of the attribute set **r**.

DCOM

DCOM supports an initial naming model of **bDmASr** with dynamic aliasing. DCOM supports location transparency and globally unique names. The extended name binding model is $\mathbf{DmA^dSeciT}$. The name management model is **bo**.

DCOM supports a distributed naming system, where a preference is placed on local resolution above remote resolution. The naming system is not structured. The extended name resolution model for DCOM is:

$$\begin{aligned}
\prec_R &: open \prec_R closed \\
\prec_P &: ambiguous \prec_P unambiguous \\
\prec_M &: exact \\
\prec_T &: out{-}of{-}date \prec_T cached \prec_T authoritative \\
\prec_L &: remote \prec_L local
\end{aligned}$$

Globe

The initial name binding model for Globe [8, 10, 176–179] is **bDMASr** with replication multiplicity and static aliasing. Names are globally unique. Globe also supports location and relocation transparency. The extended name binding model is $\mathbf{DM^rA^dSeciT}$. The name management model is **bO**.

Globe supports a tree structured directory system where resolution proceeds through nodes in the tree, starting at the client's node and progressing to the requested object's node. This behaviour introduces a recursive node preference. The name resolution model for Globe is now:

$$\begin{aligned} \prec_P &: \textit{unambiguous} \\ \prec_T &: \textit{cached} \prec_T \textit{authoritative} \\ \prec_N &: \textit{parent}-\textit{node} \prec_N \textit{local}-\textit{node} \end{aligned}$$

with the addition of the attribute set **r**.

Globus

Globus has a rich name binding model, initially defined as **BDmASR**. The inclusion of service group multiplicity, static aliasing, semantic description and contextual naming provides an extended naming model of $\mathbf{DM^sA^sSeCit}$. The name management model is $\mathbf{B^ao}$.

The name resolution model for Globus is changed through the addition of a currency preference to completely classify its support for the variation of the mutability preference. The extended name resolution for Globus is:

$$\begin{aligned} \prec_R &: \textit{open} \prec_R \textit{closed} \\ \prec_C &: \textit{static} \prec_C \textit{dynamic} \\ \prec_P &: \textit{ambiguous} \prec_P \textit{unambiguous} \\ \prec_Y &: \textit{optional} \prec_Y \textit{mandatory} \\ \prec_M &: \textit{partial} \prec_P \textit{exact} \\ \prec_T &: out-of-date \prec_T \textit{cached} \prec_T \textit{authoritative} \end{aligned}$$

with the addition of the attribute set $\mathbf{R^sf}$.

Grapevine

Grapevine supports an initial name binding model of **BDMASr** with static aliasing, active rebinding and service group multiplicity. Names are contextually structured and provide no form of location transparency or independence. The extended name binding model is $\mathbf{DM^sA^sSeCit}$. The name management model is $\mathbf{B^ao}$.

The name resolution model for Grapevine uses a locality-based resolution directory. A preference is placed on client inboxes that are local over those that are remote. The extended name resolution model is:

$$\begin{aligned}
\prec_R &: \ \mathit{open} \prec_R \mathit{closed} \\
\prec_P &: \ \mathit{ambiguous} \\
\prec_M &: \ \mathit{exact} \\
\prec_T &: \ \mathit{out{-}of{-}date} \prec_T \mathit{cached} \prec_T \mathit{authoritative} \\
\prec_L &: \ \mathit{remote} \prec_L \mathit{local}
\end{aligned}$$

with the addition of the attribute set **r**.

Infospheres

The initial name binding model for Infospheres [33, 34] is **bDmASr** with dynamic aliasing and contextual naming. The ability to limit entries to a single client provides entry control. The extended name binding model is $\mathbf{DmA^{d}SECit}$. The name management model is **bo**. The name resolution model is unchanged except for the addition of the attribute set **r**.

Java Remote Method Invocation

The initial name binding model for Java RMI is **BDmASr**. With support for active rebinding, service group multiplicity, dynamic aliasing and contextual naming the extended name binding model becomes $\mathbf{DM^{s}A^{d}SeCit}$. The name management model is $\mathbf{B^{a}o}$. The name resolution model for Java RMI (including the extension of LDAP) is unchanged except for the addition of the attribute set **r**.

Legion

Legion [73–75] has an initial name binding model of **bDmASr**. With the addition of static aliasing, semantic description and contextual naming the name binding model becomes $\mathbf{DmA^{s}SeCiT}$. The name management model for Legion is **bO**.

The name resolution model for Legion provides several stages of resolution. A client will initially attempt to resolve names using its local resolution services, proceeding to a nameserver and then onto a binding agent if local resolution fails. With the addition of the attribute set $\mathbf{R^{s}f}$, the extended name resolution model for Legion is:

$$\begin{aligned}
\prec_P &: \ \mathit{ambiguous} \prec_P \mathit{unambiguous} \\
\prec_M &: \ \mathit{partial} \prec_M \mathit{exact} \prec_M \mathit{unique} \\
\prec_T &: \ \mathit{out{-}of{-}date} \prec_T \mathit{cached} \prec_T \mathit{authoritative} \\
\prec_N &: \ \mathit{binding{-}agent} \prec_N \mathit{name{-}server} \prec_N \mathit{local}
\end{aligned}$$

5.4 The DISCWorld ORB Naming Model

This thesis proposes a transparent relocation model supported by a distributed naming service, the DISCWorld ORB system. The classification schemes developed in this Chapter are used to develop a flexible naming model for the DISCWorld ORB system; this classification can be used to compare support for transparent naming in existing systems and in the proposed system.

The naming model used within the DISCWorld ORB system must support global naming in a distributed namespace, location transparent and relocation transparent naming, and naming of both fine- and coarse-grained objects. The naming model used within the DISCWorld ORB system must also be able to define service-based aliasing and multiplicity, and descriptive attributes to support attribute-based resolution. The naming model must support naming at many different levels within the object system and represents a naming model with a combination of the requirements and facilities found in existing mobile and distributed object systems (as discussed in Chapters 2 and 4).

The mobility model used within the DISCWorld ORB system can be used to support migration at the data level through the explicit exporting of fine-grained data objects, and at the server object level through the provision of mobility APIs. The DISCWorld ORB system also supports autonomous migration through the specification of location independent itineraries; autonomous mobile objects can be created by using APIs provided by the DISCWorld ORB system. Autonomous mobile objects must be able to locate a server object through its location independent and transparent name by using the ORB system. These objects are named globally, with any context specified relative to the client that created them.

Service objects are coarse-grained objects that have globally unique names which allow them to be accessed as a service type, part of a sub-service group or uniquely through capability matching. Local names can also be provided for service objects in keeping with the characteristics of a good naming system as defined by Saltzer [158] (see page 55).

5.4.1 Application of the Naming Model

The naming system used within DISCWorld ORB system operates within a distributed directory system. Each node within the directory has its own name manager that is responsible for handling binding information for that node and any subnodes. The particular method for handling subnodes differs and is dependent on the level within the registry system in which the node resides. As has been shown in Chapters 2 and 4, naming models vary widely amongst existing mobile and distributed object systems. The DISCWorld ORB naming system has been designed to support multiple naming systems in a generic manner using the binding, management and resolution models defined in this chapter.

The implementation of the naming model mechanism has been separated from the implementation of the directory system. The naming model used can be altered at any time during an ORB systems' operation. This separation has been chosen so that different naming models, including the general naming model chosen for the DISCWorld ORB system, can be experimented with throughout a distributed object system.

To allow greater flexibility, the implementation of the naming system is performed through the implementation of complete binding, management and resolution functions. These functions take a dynamically created and alterable list of the attributes and approximation functions that the naming model must support. The ordering of these functions is given by the order in which they appear in the naming model's specification.

An implementation of the complete binding function has been developed that allows the instantiator of a registry node to specify a name binding model at run time. Specification of a required model involves defining those attributes to be included in the name binding model. An example specification can be defined as follows:

```
aliasedNames
sharedNames
multipleBinding
staticDomain
```

Each attribute that is present in the specification forms part of the name binding model. Some attributes within the classification are part of an orthogonal set; only one of the attributes can be present in a naming model at the one time to be meaningful. If no attribute within a set is present then the default operation of the DISCWorld ORB naming system is to assume the most restrictive attribute within the set. If multiple members are present then the most restrictive member amongst those specified is selected. Each of the attributes defined in the binding function set are applied to each requested binding. If any of the binding functions fail then the binding is rejected.

The set of binding functions operate on the database containing the bindings for the registry node in question. Each binding function operates in isolation and makes no direct change to the database. It is only when the binding has been deemed acceptable by the successful completion of the complete binding function that the binding is added to the database. A name binding class contains each of the name binding functions. This class can be extended to support further refinement and extension to the naming models. Due to the separation between the name binding mechanism and the remainder of the registry system no static specification of the number or type of bindings functions is required.

A similar facility is provided for the name resolution and name management functions. The model required for each function can be specified dynamically by specifying the required attributes or approximation functions. The name resolution model consists of the name resolution preference functions (the resolution and approximation functions) and the name resolution attribute functions (the complete resolution function). The preferences defined

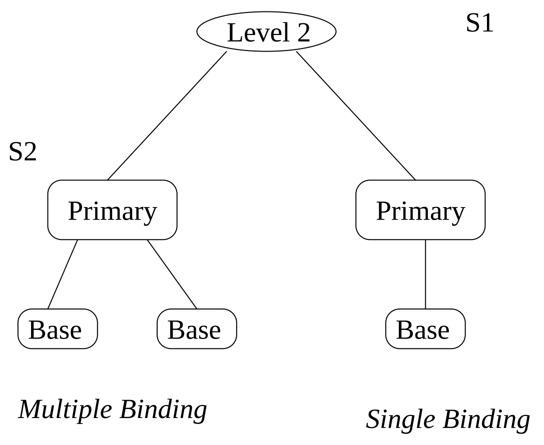


Figure 14: An example of subsystem naming models.

for a name resolution model must be ordered. The name management model consists of the mutability-based attributes and removal operations; these attributes require no ordering. The DISCWorld ORB system requires that the guaranteed preference always be present in the naming system; each server object is required to register a name, hence this attribute is a guaranteed attribute. All other preferences are optional.

The naming system of a DISCWorld ORB system can be specified on a per-node or per-domain basis. This mechanism allows different naming models to be used within the one object system, potentially allowing selective areas of a registry system to operate in seclusion while still being able to access servers resident on external DISCWorld ORB nodes. The ability to have multiple subsystem naming models can introduce conflict and coherency problems into the namespace. The mechanism is introduced to allow exploration into the interaction of naming models as represented in the integration of different object systems, and should be used carefully.

Figure 14 shows a DISCWorld ORB system where a registry domain has a different naming model to the remainder of the system. The subsystem indicated by *S1* supports single binding, while the subsystem indicated by *S2* supports multiplicity. The following two examples show potential conflicts in this system:

1. If a binding exists within *S1*, the same binding can be entered within *S2*. This potentially allows bindings in *S2* to be returned through resolution in place of the binding in *S1*.

2. If no such binding exists within *S1* but several exist in *S2*, then *S1* is unable to make its binding (a correct response). However, different resolution attempts may return different bindings from *S2*.

.

In these examples inconsistent bindings can be entered and accessed while maintaining the validity of the naming model. These examples also show that it is possible to define subsystem naming models that interfere with the naming models of other subsystems. This has the potential to break the expected semantics of such systems as shown in the second example. These effects must be considered when defining, or changing, subsystem naming models. For example, it is possible to have subsystems with more flexible multiple binding attributes, however these bindings need to be defined as local only. These bindings can then not be used to respond to remote requests for resolution and can not interfere with the multiple binding attributes defined in other subsystems.

Given a set of name binding, resolution and management functions, the level of coherency checking is defined by the mixture of global knowledge and uniqueness required in the system. A system that requires globally unique names that can be reused requires full coherency checking; the importance of coherency checking is defined by the scale of the system and any response requirements that must be supported.

5.4.2 The General Naming Model

As defined in Section 5.4.1 the naming model used within the DISCWorld ORB system can be defined on the basis of a node or a subset of nodes. A default naming model is defined here for use as the general adopted model for naming within the DISCWorld ORB system.

The general name binding model for the DISCWorld ORB system exhibits the properties of dynamic aliasing, replication multiplicity, location independence and transparency. Names are globally unique with aliases able to be defined on a local basis. Entries can be restricted at the request of the object in a similar fashion to the mechanism found in Infospheres. The name binding model is $\mathbf{DM^{r}A^{d}SEcIT}$.

The general name management model exhibits active rebinding as its form of mutability and supports relocation transparency. The name management is $\mathbf{B^{a}O}$.

Name resolution is performed through a hierarchical distributed directory system with cached information. Attributes can be specified as optional or mandatory and the database contains guaranteed information. Some information is unambiguously matched to an object while other information may match multiple objects. Using the resolution mechanism it is possible to receive a partial match, an exact match or a unique match. Some information is time sensitive; information may be altered dynamically with some aspects (such as client requirements) given a time-to-live value. The general name resolution for the DISCWorld

ORB system is defined by:

$$\begin{aligned}
\prec_R &: \textit{open} \prec_R \textit{closed} \\
\prec_P &: \textit{ambiguous} \prec_P \textit{unambiguous} \\
\prec_U &: \textit{static} \prec_U \textit{dynamic} \\
\prec_M &: \textit{partial} \prec_M \textit{exact} \prec_M \textit{unique} \\
\prec_T &: \textit{cached} \prec_T \textit{authoritative} \\
\prec_N &: \textit{higher}-\textit{level} \prec_N \textit{primary} \prec_N \textit{local}
\end{aligned}$$

with the addition of the attribute set $\mathbf{R^sF}$.

5.4.3 Lifecycle of DISCWorld Names

The processes of binding, management and resolution within the DISCWorld ORB system are multi-stage processes dependent on the current state of the name or reference. Each name has a lifecycle that defines what capabilities it has at any one time. These capabilities control what operations and attributes present within the naming model are applied to operations instigated by the client in receipt of the name.

A summary of the naming lifecycle is as follows. A client requests resolution from the DISCWorld ORB system by specifying a name. If this name can be resolved by the ORB system, a reference is returned to the client; initially, this reference is not connected to the referenced object. Upon the first use of the reference, connection is made and is maintained until either the client closes the connection (by deleting the reference) or the referenced object fails. Connection is maintained through object migration. These stages of the naming lifecycle are discussed further below.

Initially a client requests an object by defining a name which represents an object or a group of objects. Examples of the kinds of names that can be used by a client are an attribute, a service name, a well known name or a *token*. A token is used once a name has already been resolved; this process of token resolution and the usage of tokens within the system is examined at a later stage within the name lifecycle.

The client requests resolution and management of a name by contacting a directory node; access to directory and naming services is provided through a local or remote ORB. This ORB provides access to the distributed registry service which allows resolution for objects anywhere within the registry system. The process of resolution transforms the reference (being the name) from the unknown state to the unconnected state.

An object is selected by the directory system that matches the name. This can be either the exact service, one of a service group, or a service that matches an attribute description. A client interface fragment is returned to the client to represent the second stage of the reference: the unconnected reference. This fragment contains a location hint that can be used to connect with the service object. At this point, no unique capability or token is provided as there is no direct binding between the objects.

Once the client accesses the service object through the reference (by invoking a method or performing some task on the service) the reference progresses to the connected state and a token is provided by the service object. This token acts as a capability to the unique name of the service and a capability to any stored invocation queues during a migration. This token can be used at a later point to request a binding to the currently accessed service object. A token uniquely identifies a service object by defining its service group or other naming which can be used later to optimise resolution. It also uniquely identifies an object and the channel connecting the client to the service. A token contains the following information:

- service group name,
- optional alias name,
- optional attribute specification,
- unique identifier, and
- channel identifier.

The optional names are provided only as hints as this information can change during the lifetime of the service object. By using these hints, resolution can be directed at those parts of the directory in which the service object is most likely to be held.

A connected reference is then transformed to the channel state. When a reference transforms into a channel state it causes a migration fragment to be created on the service object side. This migration fragment is responsible for handling invocations and accesses on the service object and is a representative of the remote end of the connecting channel. A depiction of the lifecycle of a DISCWorld ORB reference is shown in Figure 15.

References can move between the unconnected, connected and channel states at any point during the lifetime of the reference. A reference may move from the channel state to the unconnected state through connection loss due to channel or service failure. The client interface fragment can then use its cached location hints to attempt to reestablish the connection or it can access the directory for updated resolution using the token as the name to be resolved.

These different states of a reference have different properties. An unknown reference may be shared, is mutable and is both transparent and independent. It does not represent any binding between two objects but the signature of the binding to take place. An unconnected reference is also location transparent, but does contain a location hint causing the reference to be partially location dependent. These references can also be shared, are mutable and may act through an aliased name.

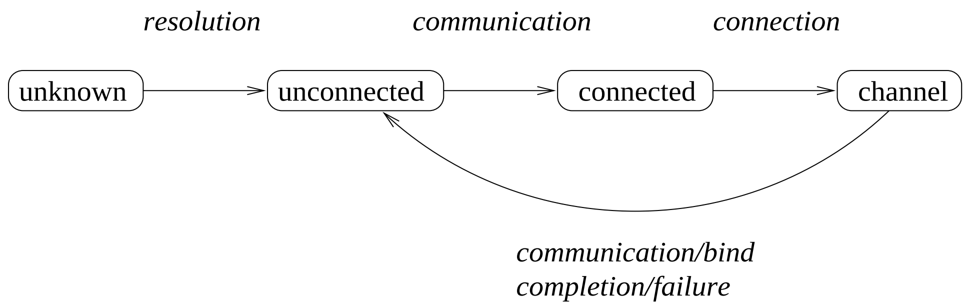


Figure 15: Lifecycle of a DISCWorld ORB reference.

System	Binding	Management	Resolution
d'Agents	$\mathbf{DmA^{d}SeCit}$	**bo**	**r**
Obliq	$\mathbf{DmA^{d}SecIT}$	**bO**	**r**
Infospheres	$\mathbf{DmA^{d}SECit}$	**bo**	**r**
Legion	$\mathbf{DmA^{s}SeCit}$	**bO**	$\mathbf{R^{s}f}$

Table 6: Differences in classification under the extended naming model.

A connected reference represents a shared name which is transparent but partially location dependent. These references are also mutable (as the connection is to the migration fragment) and are not unique. A channel reference is where the model of reference makes its most dramatic change. These references are globally unique and can not be shared. The reference is mutable and may be accessed through an alias.

5.5 Summary

The name binding and name resolution models described in Chapters 3 and 4 are refined and extended in order to accurately classify mobile and distributed object systems with a requirement for transparency. This classification introduces classifications for information previously unconsidered by Bayerdorffer [15, 16] and Bowman *et al* [26]. The extended attributes and preferences defined for these models are further divided into name binding, name management and name resolution categories. Reclassification of the existing mobile and distributed object systems examined in Chapters 2 and 4 is also performed.

Table 6 shows the reclassification of a subset of the systems examined in this thesis. According to the existing classification models, all of these systems share the same name binding classification of **bDmASr**. Table 6 shows that there are distinct differences in the name binding models of these systems as indicated by their classification under the extended naming model.

Tables 7, 8, 9 and 10 present a summary of the support for the extended naming model described in this chapter. For each of the tables, a $\surd$ indicates that the system in question supports the specified attribute, while a $\times$ indicates that the system supports the most restrictive orthogonal member of the attribute. For example, a $\times$ for the multiplicity indicates that the system supports static binding.

The extensions to the naming models encompass issues present in current distributed object systems, and issues particularly specific to object systems intended to support transparent migration. Two forms of the extended naming model can then be determined: one that can be applied generally to object systems and one that can be applied to systems where location and relocation issues are of importance. The removal of the locality-based preferences and the location and relocation-based attributes from the extended naming model enable the model to be generally applied to object systems. This restricted naming model can then be applied to current object systems where object migration is not a factor.

The DISCWorld ORB naming model is introduced and formally defined. The DISCWorld ORB naming system is designed to operate over a hierarchical distributed directory system where different name models may be in use. Hence a generic name model integration system, based on the use of the complete binding, complete management and complete resolution functions, is introduced. A general naming model for use in the DISCWorld ORB system and the effects of the lifecycle process of names is also described.

The extended naming model, with its formal definition can be used to define a generic naming system separating the implementation of system dependent factors from general naming model issues. The extended naming model can be used to define both object systems and object systems requiring transparency and migration.

System	D	M	A	S	E	C	I	T
Ajents	$\surd$	$\times$	A^d	$\surd$	$\times$	$\times$	$\times$	$\times$
Aleph	$\times$	M^s	$\times$	$\times$	$\surd$	$\times$	$\times$	$\surd$
CORBA	$\surd$	$\times$	A^d	$\surd$	$\surd$	$\times$	$\times$	$\surd$
d'Agents	$\surd$	$\times$	A^d	$\surd$	$\times$	$\surd$	$\times$	$\times$
DCOM	$\surd$	$\times$	A^d	$\surd$	$\times$	$\times$	$\times$	$\surd$
DISCWorld	$\surd$	M^r	A^d	$\surd$	$\surd$	$\times$	$\surd$	$\surd$
Emerald	$\surd$	$\times$	A^d	$\surd$	$\times$	$\times$	$\surd$	$\surd$
Globe	$\surd$	M^r	A^d	$\surd$	$\times$	$\times$	$\times$	$\surd$
Globus	$\surd$	M^s	A^s	$\surd$	$\times$	$\surd$	$\times$	$\times$
Grapevine	$\surd$	M^s	A^s	$\surd$	$\times$	$\surd$	$\times$	$\times$
Infospheres	$\surd$	$\times$	A^d	$\surd$	$\surd$	$\surd$	$\times$	$\times$
Java RMI	$\surd$	M^s	A^d	$\surd$	$\times$	$\surd$	$\times$	$\times$
Legion	$\surd$	$\times$	A^s	$\surd$	$\times$	$\surd$	$\times$	$\surd$
MOA	$\surd$	$\times$	A^s	$\surd$	$\times$	$\times$	$\times$	$\times$
Mobile Process	$\surd$	$\times$	$\times$	$\surd$	$\times$	$\times$	$\times$	$\surd$
Obliq	$\surd$	$\times$	A^d	$\surd$	$\times$	$\times$	$\surd$	$\surd$
Regional Directories	$\times$	$\times$	A^d	$\surd$	$\times$	$\surd$	$\times$	$\times$

Table 7: Existing object system support for the extended name binding model.

System	R	F
Aleph	R^b	$\surd$
DISCWorld	R^s	$\surd$
Globus	R^s	$\times$
Legion	R^s	$\times$

Table 8: Existing object system support for descriptive and attribute-based naming.

System	B	O
Ajents	B^a	$\times$
Aleph	B^a	$\surd$
CORBA	$\times$	$\times$
d'Agents	$\times$	$\times$
DCOM	$\times$	$\times$
DISCWorld	B^a	$\surd$
Emerald	B^r	$\surd$
Globe	$\times$	$\surd$
Globus	B^a	$\times$
Grapevine	B^a	$\times$
Infospheres	$\times$	$\times$
Java RMI	B^a	$\times$
Legion	$\times$	$\surd$
MOA	B^a	$\surd$
Mobile Process	B^r	$\surd$
Obliq	$\times$	$\surd$
Regional Directories	B^a	$\surd$

Table 9: Existing object system support for the extended name management model.

System	Locality	Node
CORBA	$\surd$	$\times$
d'Agents	$\surd$	$\times$
DCOM	$\surd$	$\times$
DISCWorld	$\times$	$\surd$
Globe	$\times$	$\surd$
Grapevine	$\surd$	$\times$
Legion	$\times$	$\surd$
MOA	$\times$	$\surd$
Obliq	$\surd$	$\times$
Regional Directories	$\times$	$\surd$

Table 10: Existing object system support for the extended name resolution model.

Chapter 6

The DISCWorld ORB System

The DISCWorld metacomputing environment [78,82] is a distributed object system designed to support integration of independently implemented clients and servers. DISCWorld provides services for use in a geographical information system (GIS) and has been used to support several distributed high performance research projects [37, 38, 103].

Client and server objects interact within the DISCWorld metacomputing system through the use of enabling services. These services provide scheduling of requested operations, service reconfiguration, the location and monitoring of server objects and the ability to perform bulk data transfer and storage. Scheduling within DISCWorld is described in [79, 97], bulk data transfer and storage in [144, 145], and service reconfiguration in [80]. The namespace representing server objects integrated into the DISCWorld system is managed by the DISCWorld ORB system [51, 104], which provides server object location, relocation and monitoring mechanisms. The DISCWorld ORB system additionally provides support for service mobility and replication. The ORB system is designed to support attribute-based service selection and transparent name resolution. These facilities remove the need for services and clients to explicitly program their own communication and migration requirements.

Figure 16 shows an abstract view of the DISCWorld system, where the client views the DISCWorld system as a single entity or cloud accessible through WWW interfaces. The communications and service brokerage infrastructure are kept separate from the client interface. A client accesses the DISCWorld system through an abstract interface. Through this interface a user query is injected into the system, in the form of one or more data entities and a sequence of operations to be applied to them.

The DISCWorld model of computation involves several steps with the aim of breaking up potentially complex client queries into components that can be matched against services existing within the DISCWorld system. The client can specify the required services through an attribute set, which can then be matched against the attributes provided by each registered service. Services are also selected upon cost estimation; the ability to select an

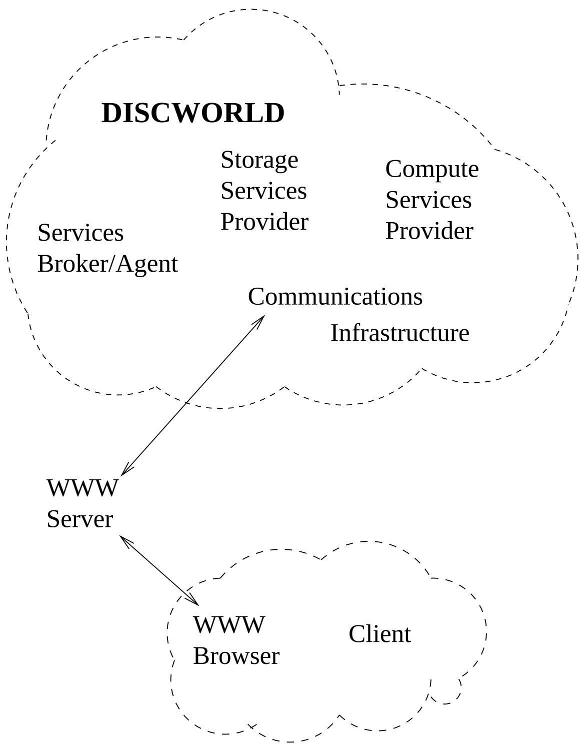

DISCWORLD
Storage Services Provider
Compute Services Provider
Services Broker/Agent
Communications Infrastructure
WWW Server
WWW Browser
Client

Figure 16: Abstract view of the DISCWorld system.

appropriate service according to what the client is willing to pay is considered important. The data entities need to be appropriately marshalled, and the services brokered to obtain a result that is finally delivered to the user.

In practise, the kinds of data objects transported throughout DISCWorld are large image data and bulk repository data. To support efficient transfer of large data entities, specialised bulk data transfer services are provided within the DISCWorld metacomputing environment.

It can be seen that the potential requirements to manage a service within DISCWorld are:

- location of services,
- activation of services,
- searchable indexes of services,
- updatable indexes of services (dynamic domain),
- location of registry nodes, and
- access to registry nodes from any machine type.

These requirements are met by the integration of a distributed ORB system that supports a dynamic domain. Each ORB can be used as a local access point to a distributed, location independent and transparent global namespace. Heterogeneity is provided by implementing the ORB system in a platform independent language.

The contribution of this thesis towards the DISCWorld system is service management using the DISCWorld ORB system. This includes the development of a relocation transparent naming model, an exploration of the potential scalability of this model and the implementation the DISCWorld ORB system designed to support this model and scalable, distributed name resolution. Additionally, implementations of client and server application code to expliot the DISCWorld ORB system have been developed.

The object model used in the DISCWorld system is similar to that used in Emerald [102]; objects may be fine-grained data elements or they may be coarse-grained objects. This object model allows both kinds of objects to be bound, managed and resolved according to the specified naming model. Objects exist within a global namespace with an adaptable naming model as described in Section 5.4. Object migration is directly supported, both in terms of specialised communications models that take advantage of mobility, and service and data object migration.

Transparent object location is supported through the use of a distributed directory system. The distributed directory system is organised as a hierarchically structured tree. This structure is adaptable and fault resilient, as it can reorganise itself in an attempt to overcome partial node failure. The adaptiveness of the system and the models of fault resilience supported are described in more detail in Section 6.2.3. The directory system has also been designed to be transparent in both its structure and initialisation; nodes within the directory system are capable of operating at any level within the hierarchy and are capable of changing their position within the hierarchy without user intervention. Object information is lazily dispersed and cached throughout the directory structure in response to information queries and requests. Information that is lazily dispersed is sent to other nodes only as a response to queries.

Transparent relocation is of great importance in a dynamic object system; it is essential that connections are maintained regardless of the location, or the current migration state, of the service object. Transparent relocation is supported through an explicit update method with a backup mechanism utilising the distributed directory service. No home location or forwarding location mechanisms are used, requiring no additional or residual code objects within the system.

This chapter describes the design and operation of the DISCWorld ORB system and its integration with the DISCWorld metacomputing environment. In Section 6.1 example applications, both legacy and native, are presented. The use of mobility and replication support are also described with respect to the type of applications supported by the DISCWorld system.

Section 6.2 discusses the DISCWorld ORB system in more detail along with the operation of the distributed directory system. The generic and adaptable implementation of the naming model is described. The transparent location and relocation mechanisms employed in the DISCWorld ORB system are examined in Section 6.3. Detail is provided for the structure of names and references as used within the system.

The APIs and communications models provided by the DISCWorld ORB system are discussed in Sections 6.5 and 6.6 respectively.

6.1 Example Applications

The DISCWorld system is designed to support both legacy applications and purpose-built migratable services. The inclusion of legacy applications and applications written in high performance, non-migratable languages is undertaken to support research activities in the areas of image management and geographical information systems.

6.1.1 Example Legacy Applications

The fusion of large remotely sensed datasets with ground-truthing, directly observed data is an important but computationally costly challenge. Satellite imagery data generally represents a data gridding pattern that is completely different from ground truth data. A typical instance of this problem arises for the Soil and Land Management Cooperative Research Centre in its project to investigate correlations between vegetative growth cover and rainfall conditions at high ground resolution. Data sources for rain prediction purposes are from the GMS5 geostationary meteorological satellite and the NOAA polar orbiting satellite. Ground truthing data is available in the form of the Australian "Rainman" data set, representing Bureau of Meteorology ground station measurements of rainfall for the points throughout Australia.

To carry out a useful comparison between multiple datasets it is necessary to have techniques to interpolate spatial data from one set of arbitrary sample points to another, possibly gridded, layout. One technique for tackling this is known as kriging [103, 141] and can be formulated as a matrix problem connecting the datasets. The kriging problem is an interesting example of a computationally intensive processing component in a GIS. Many of the spatial data manipulation and visualisation operations in a GIS may be relatively lightweight and capable of running on a PC or a low performance computing platform, but kriging a large data set is too computationally demanding and therefore needs to be run on a separate "accelerator platform". Implementations of kriging interpolation have been developed for a 128-node Connection Machine CM5 and a farm of AlphaStations; these implementations have been developed to support high performance interpolation. These implementations exist as legacy applications that have been integrated into the DISCWorld environment.

The DISCWorld metacomputing system enables a client to select the most appropriate service that is capable of performing the requested task. The client's restrictions, such as the required cost, performance or location of the service, can be used to select an appropriate service instance.

Interesting issues in the construction of such a system include the tradeoffs between the communications cost of transferring data between clients and servers compared to the speed advantages of using a powerful computational server instead of running the kriging on the client. It is also possible to allow the client application to choose the computational accelerator for the task required. There may be a monetary difference between fast, medium and slow servers, all of which are functionally capable of providing the required service. Other issues include the network reachability of the servers and the estimated time to deliver the solution compared to the end-user requirements. The problem complexity also varies based on other pre-processing and post-processing activities as required by the end-user.

Figure 17 shows the process of legacy object integration into the DISCWorld metacomputing environment. Services to support the kriging operation through interpolation, dataset integration and classification are registered with the DISCWorld ORB system. The DISCWorld processing manager provides access to storage services and schedules the operations on the registered interpolation service objects.

Access to the kriging service and additional data management and transfer services are provided through the abstraction of the WWW-based interface. Legacy applications can be registered through the use of specialised APIs provided by the DISCWorld ORB system that allow attribute specification but do not provide any support for mobility or replication due to the unknown nature of the legacy object.

6.1.2 Example Native Applications

There are several example applications constructed solely for the DISCWorld system, some of which exploit mobility and policy specification in their usage. DISCWorld supports its own communications and scheduling protocol constructed around the DISCWorld Remote Access Mechanism (DRAM) [81,98], which allows services with a single purpose (operation) to be combined and operated remotely. DRAMs act as remote references to services or to data and can be moved abstractly around the DISCWorld system while maintaining their referential integrity.

DISCWorld services can be accessed through either DRAM references or named DISCWorld ORB references. DISCWorld ORB references allow services to provide more than one operation or function by specifying multiple interfaces that can be supported through the fragmented object model. When accessing a service as an ORB service, the

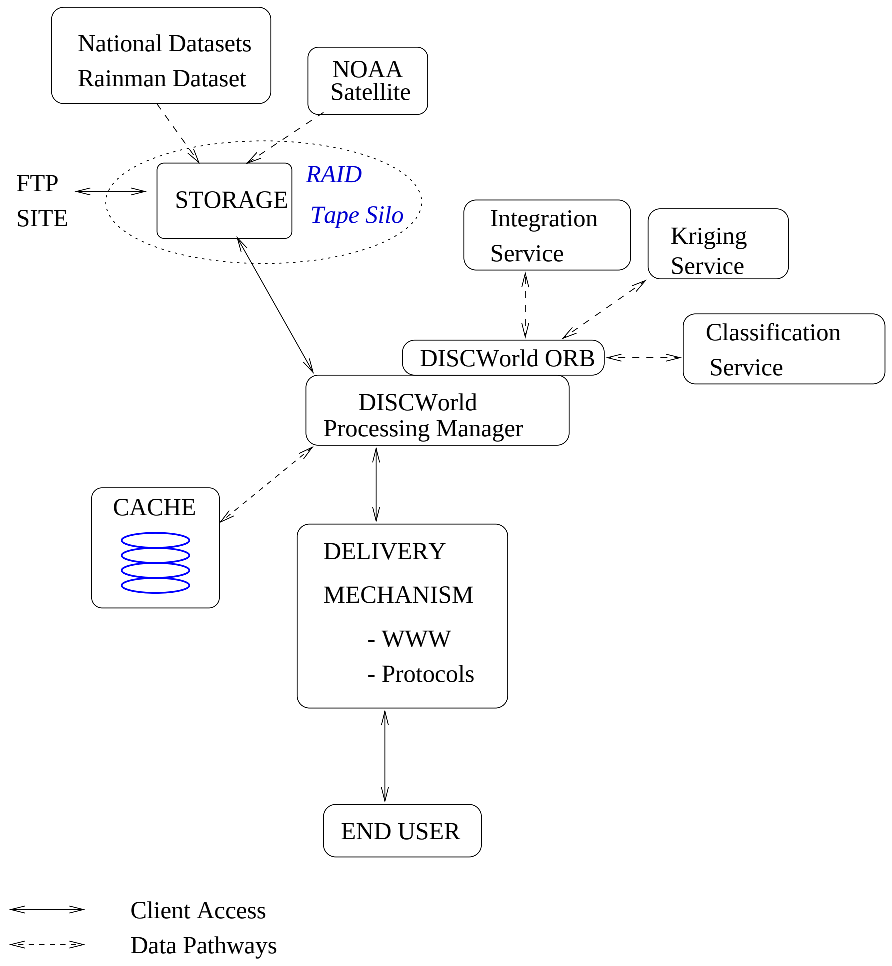


Figure 17: Integration of legacy applications into DISCWorld.

DISCWorld ORB system can be used to provide naming, replication, mobility and cloning support services.

An example of a native DISCWorld ORB application is the DISCWorld ORB distributed web server. The web server has been designed so that it contains two service objects: a front-end accessible by clients of the web server, and multiple distributed back-end service objects that provide that actual serving. The back-end servers are cloned and distributed throughout the registry domain by the DISCWorld ORB system in order to provide as many distributed web server nodes as possible within the domain.

Native DISCWorld services are registered within the DISCWorld system and have full access to the mobility, relocation and replication services provided by the ORB system. To access these services clients can interact with the DISCWorld system through WWW-based interfaces or through direct, named references. This process is shown in Figure 18.

The front-end exists to forward incoming requests to a selected back-end server. The front-end server is able to update its list of back-end servers dynamically as more servers are

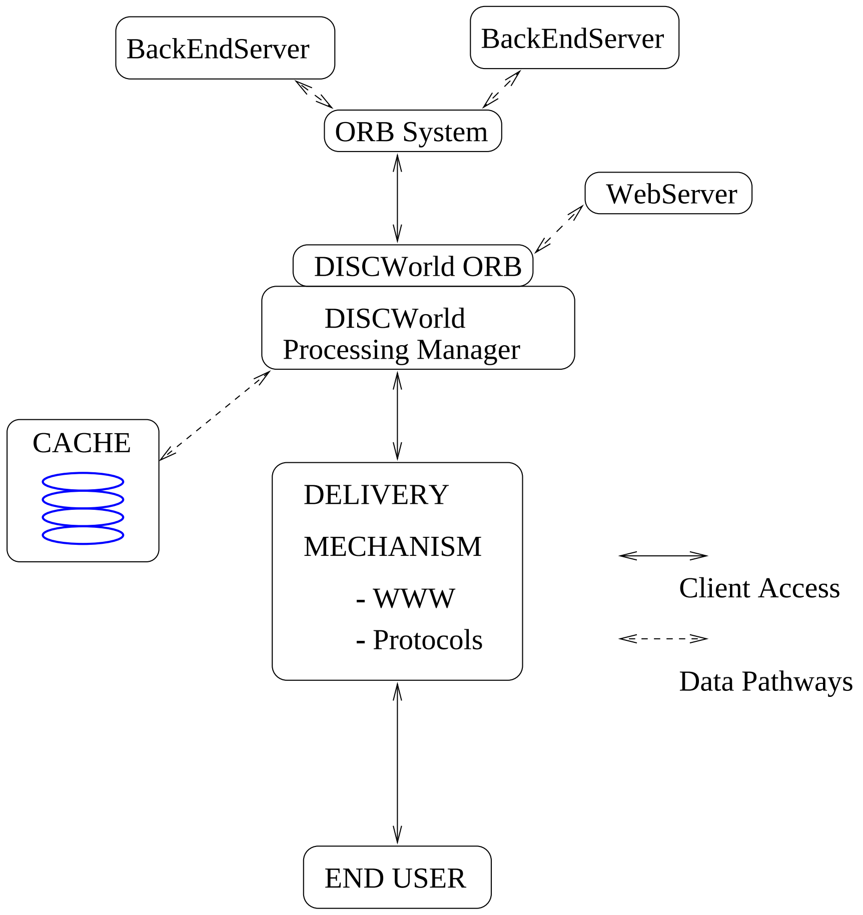


Figure 18: Integration of native applications into DISCWorld.

added or the distribution changes. Initialisation of a front-end server creates a single object on a well known port that is able to receive hypertext transfer protocol (http) requests from clients. When the front-end server is created it contacts an ORB and requests a list of all back-end servers within the DISCWorld ORB system. This request is made by specifying an attribute set containing the base name of the class hierarchy defining the back-end server.

Figure 19 shows an example of the process of web server initialisation. Figure 19 (i) shows an object system where four back-end servers have registered with a local ORB. The registration information for these servers is maintained in the ORB's database segment, indicated by *db*. In (ii) a front-end server, indicated by FE, requests a list of matching back-end servers from a local ORB. The ORB then returns the matching list of all the back-end servers, indicated by BE, that have been registered within the ORB system. This list consists of a set of location independent references (client interface fragments) with cached location hints. In (iii) the front-end server then uses the list of references to connect to each of the back-end servers.

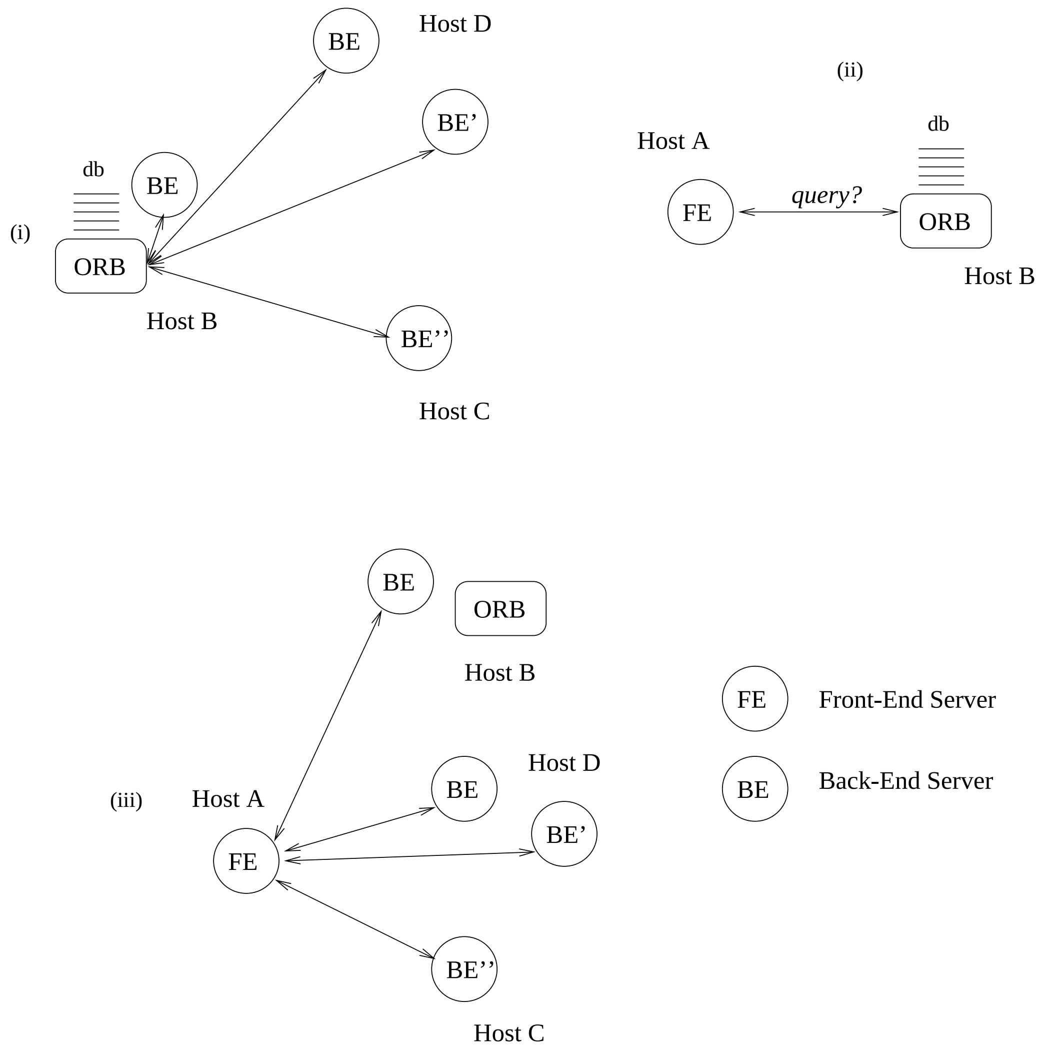


Figure 19: Webserver integration using the DISCWorld ORB model.

The back-end servers have been designed to support extension and allow expansion of protocol knowledge though subclassing. A default back-end server has been designed that is capable of supporting several common file formats including hypertext markup language (html), plain text, and postscript. Extensions to this support can be implemented by extending the back-end server class to support additional file formats, such as language specific formats. Service group multiplicity and dynamic aliasing are used to link all implementations of the back-end server to the same name which can then be resolved by the front-end server.

At any point within the lifetime of the front-end object, the set of back-end servers available can be updated. Transparent relocation enables any of the servers within the web server system to migrate without losing future or current connections with either clients or other web server components.

6.2 The DISCWorld ORB System

The DISCWorld ORB system consists of a network of ORB nodes that form a hierarchical directory structure. ORBs can be dynamically integrated into the system and can adapt to interact at any level within the hierarchy. The DISCWorld ORB structure provides a distributed, global namespace through which name binding, management and resolution can be performed. These facilities enable the transparent location and relocation of objects within the system.

6.2.1 ORB Model

The ORB model defined in the DISCWorld ORB system is similar to that defined in the CORBA specification [18, 136, 138]. An ORB is used as an intermediate third party through which clients can obtain references to servers, and servers can register themselves and define attributes. An ORB within the DISCWorld ORB system can be viewed as a directory providing access to naming services, migration and replication facilities. Each ORB provides an API through which clients and servers access its services. For example, interfaces are provided to perform name binding, resolution and management, and attribute-based searching for both native DISCWorld ORB objects and legacy objects.

ORBs within the DISCWorld ORB system act as access points to the directory and naming services provided by the system. Through registering with a DISCWorld ORB node, a service object can advertise its services and have its references resolved throughout the ORB system; clients are able to access any server registered within the system. These references are transparently updated upon service migration or failure. The ORB system provides additional functionality beyond that of an access point to the registry or naming services. This additional functionality includes support for object replication, cloning and migration. These services can be activated and controlled through policy specification on a per-service basis. Integration of services and policy definition is described in more detail in Section 6.5.1.

The replication support system allows a service to be replicated and to have the option of distributing these replicas throughout the registry system. Each replica receives the same messages and invocation requests as the original or parent object. Replication can be used to provide some fault tolerance as the service is capable of surviving partial node failure. The cloning support system allows a service to be cloned and to have the option of distributing these clones throughout the registry system. Each clone acts independently and can be used to support service-group multiplicity.

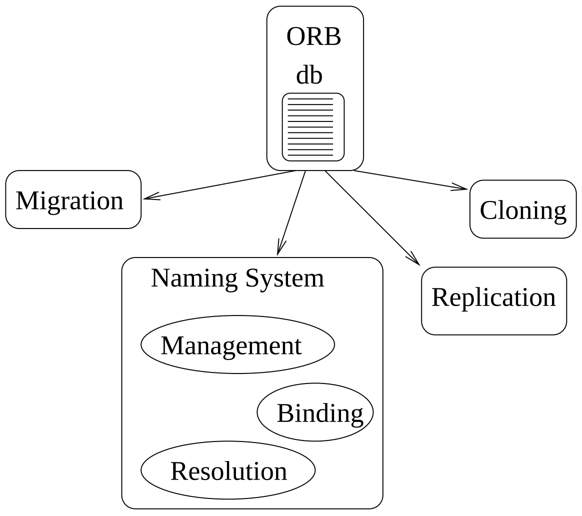


Figure 20: Component model for a single registry system.

Mobility is supported by allowing an object or the registry system itself to request relocation. The mobility support system provides mechanisms to dynamically create mobile objects which can be used to travel around a registry domain locally accessing its services. The ORB-based communications model used within the DISCWorld ORB system relies on the attachment of mobile fragments to each server registered with the ORB. Mobile fragments are similar in concept to the scion used in stub-scions and adapters used in CORBA. What distinguishes mobile fragments from scions or skeletons is that mobile fragments exist for only those references that are connected (unlike a scion model) and they provide facilities for object migration and transparent relocation. Migration fragments do not act as residual code, they are only present with registered service objects.

Essentially an ORB within the DISCWorld ORB system exists as a directory structure with several contact points for additional services, as shown in Figure 20. In the component model shown in Figure 20, an ORB maintains a segment of the global namespace database and has access to the remainder of the database through contact with other ORB nodes. Additional modules are responsible for managing the naming system (including name resolution, management and binding), object migration, replication and cloning. These modules act as ORB-based services that can be utilised by client and server objects within the ORB system. The migration service object is used as a arrival point for mobile objects when they migrate between nodes. The migration access point is responsible for registering the object with a suitable policy object, potentially defined by the mobile object itself.

The DISCWorld ORB model is designed for extensibility; more objects with specific purposes can be added to the ORB system without requiring a specific interface or interaction with the ORB or directory itself.

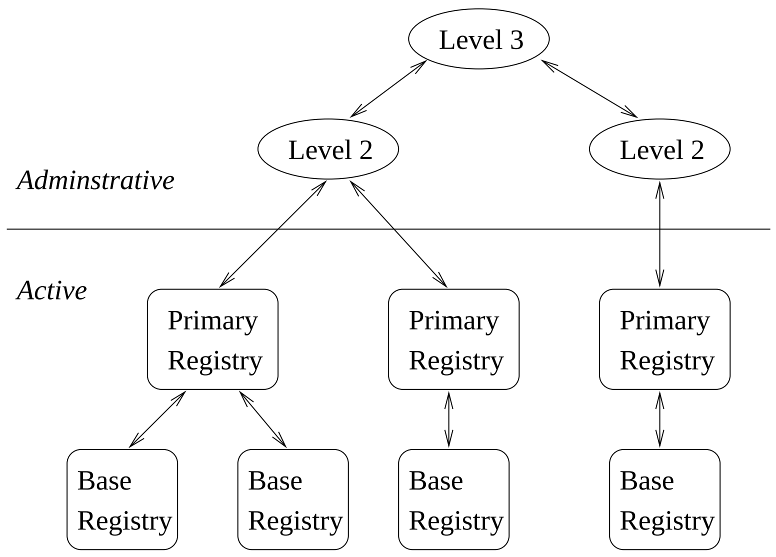


Figure 21: Structure of a DISCWorld ORB system.

6.2.2 Distribution Model

Each ORB within the DISCWorld ORB system exists as part of a hierarchically structured, non-replicated distributed ORB system. The structure is made up of multiple levels of ORBs that cooperate to respond to client and server requests. The global namespace is distributed throughout the system with each ORB node responsible for those parts of the database relevant to services registered at itself and at any of its leaf nodes. A single ORB is responsible for responding to incoming requests, forwarding unsatisfied requests to parent nodes and managing the part of the database under its care.

The DISCWorld ORB hierarchy is divided into two levels: the administrative level and the active level. ORBs within the active layer are able to respond to client and server object requests as well as requests from other ORBs within the system; ORBs within the administrative layer are only able to respond to requests from other ORBs. The distributed ORB system is divided into these two layers to form a separation of ORBs who are contacted directly by client and server objects, and ORBs that are responsible for performing remote name binding and name resolution tasks.

Figure 21 shows an example of a distributed DISCWorld ORB structure. Nodes within the active layer include base level registries and primary registries; each primary registry is responsible for managing a subgroup of base level registries and for communicating requests for remote resolution to higher level administrative nodes. Each node is able to change its position within the hierarchy through adaption. Nodes within the active layer are termed level 1 registries.

The group of registries managed by a primary registry is termed its *registry domain.* A registry domain consists of a primary registry and all base registries within the specified contact distance. This can be defined formally as the set of registries within a latency n of the primary registry, where the latency is defined by the response time in querying another registry. This value is defined by the creator of the registry domain and may differ between domains within the system.

The selection of a primary registry based on its round-trip response time (depending on the latency in the connecting link) is an example of a network probe, a commonly used technique for resource location [77]. An example of a network probe technique is *traceroute* where packets with increasing "time-to-live" values are broadcast to gather host information within different bounds. In this example, "time-to-live" values are defined as the number of hops. The mechanism used in the DISCWorld ORB system is also similar to a *directed broadcast* as defined by Boggs [23]. A directed broadcast enables a broadcast to be sent a particular network. Extensions to the work, such as *anycast* [143], provide more efficient resource location mechanisms but do not support selection of services based on other attributes. As with other distributed object systems, the focus in the DISCWorld ORB system is on LAN-based location rather than network topology [2, 77].

The information flow throughout the directory system is shown in Figure 22. Part (i) of Figure 22 shows a single registry domain consisting of two base level registries and a single primary registry. When a server registers or deregisters at a base level registry, the registration information is forwarded to the primary registry; the primary registry has a complete image of the services contained within its domain. By providing access to the global namespace through any level 1 registry, communication can be distributed throughout the registry system removing the potential for central points of failure or bottlenecking. Part (ii) of Figure 22 shows a multilevel registry system, where primary registries communicate with a second level registry used to combine multiple registry domains.

Instead of a push model, where all information is forwarded as it changes, as is found in a single domain system, a multiple domain system uses a pull model to gather information. For example, if a request comes up through a single domain system to a second level registry for name resolution which can not be resolved, all primary registries known to the second level registry are polled for updated lists of their services and name bindings.

Figure 23 shows the information flow through a domain registry system in response to a client's name resolution request. Figure 23 (i) shows a single domain system with two base registries. In (ii) a client requests name resolution for the object C through its local ORB, which in turn forwards the request to its primary registry. In this example the primary registry is able to resolve the request and return a reference to object C resident within its domain. In (iii) the client connects its reference to the object C.

By caching information gathered through responses to queries, an administrative registry is able to poll those nodes that previously responded successfully ahead of other nodes within the system. This preference enables names to be resolved quickly if the requested object is

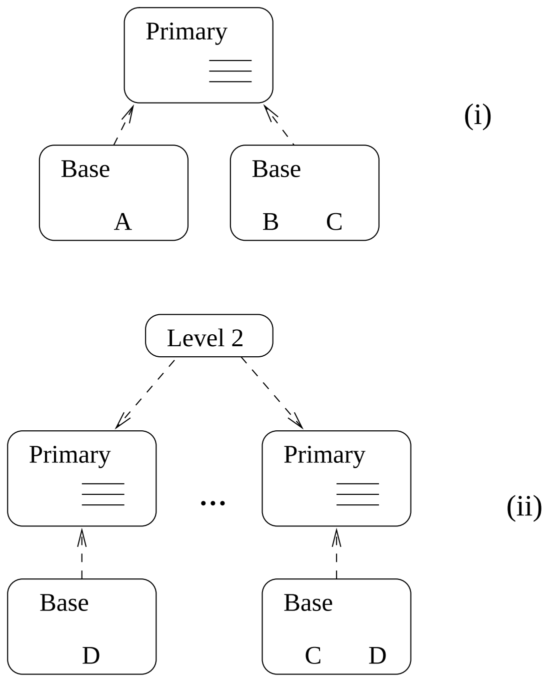


Figure 22: Information flow through the ORB system.

still at its previous location. No cached information is explicitly trusted but it is used as a hint to specialise requests. This policy is based on the theory of locality [174] that specifies that it is more probable that an object will relocate to a nearby host than to a far host. In the case of local migration, there is a likelihood that the mobile object is within the same registry domain and hence is known by the same primary registry as before the migration.

The combination of information push and pull models allows the registry system to expand without dramatically increasing the information flow required to maintain a fully replicated registry system. Information is only requested in the case of client need, with information being lazily distributed throughout the registry system.

6.2.3 Adaptability

A registry domain consists of a primary registry and several base registry ORBs. This model immediately appears to be flawed as the primary registry acts as a central point of failure. This situation is similar to a home location model, where a single object maintains all reference mappings. The difference is that, in an ORB-based system, the object can be accessed through several distributed entry points: the base level registries. Scalability in this form of system increased but the failure points are identical. To overcome this problem each DISCWorld ORB is capable of assuming the role of primary registry within its registry domain with some knowledge loss.

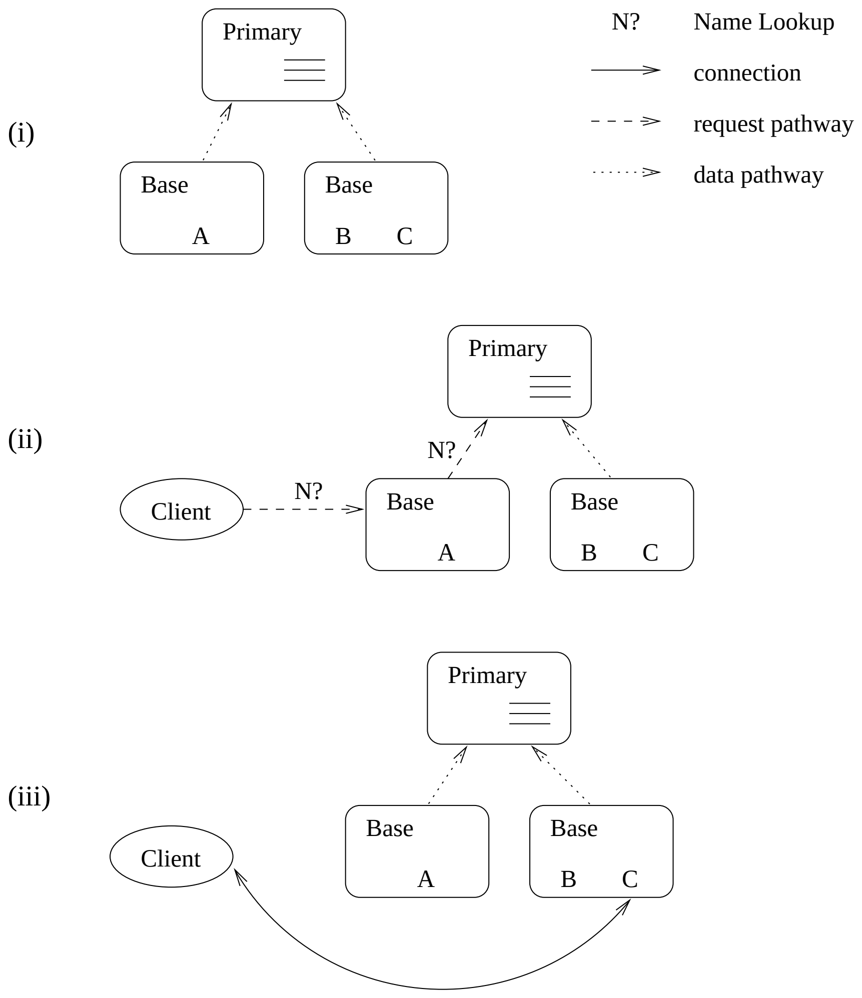


Figure 23: Distributed multiple registry location model.

When an active layer ORB is created it first attempts to locate a primary registry within the appropriate response latency n. If possible, the ORB joins the registry domain as a base registry; if not, it creates itself as a primary registry so that future ORBs within a domain distance can join the newly created domain.

Each ORB is capable of dynamically adapting its behaviour in response to external failures. Adaptiveness is instigated in a lazy manner; if a registry detects that its current primary registry is no longer accepting connections, it performs a broadcast location for any other existing primary registries within its registry domain. If the registry locates another primary registry it joins the found registry domain and forwards the list of services registered with it. If a registry is unable to locate an existing primary registry it assumes the position of primary registry. The registry is now able to accept primary registry requests and forwarded registrations from other registries within the domain. This process is shown in Figure 24.

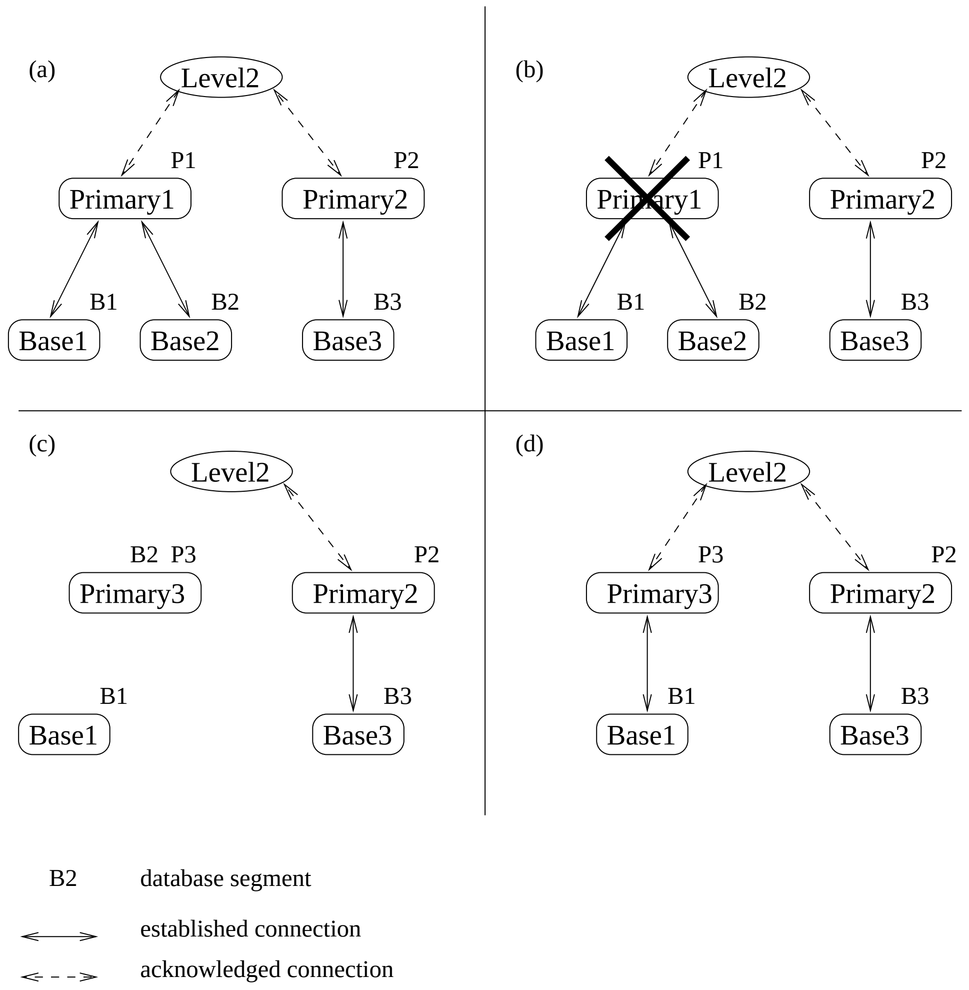


Figure 24: Maintaining domain structure through adaption.

Figure 24 (a) shows an ORB system consisting of two registry domains linked by a level 2 registry. In (b) the primary registry indicated by *primary1* fails. In response to the failure (discovered either through a failed request or binding), the base registry indicated by *base2* attempts to locate another primary registry. If this fails then *base2* becomes the new primary registry, *primary3*, managing the segment previously known to it through its *base2* incarnation and cached information from the level 2 registry. This is shown in Figure 24 (c). Existing connections between the failed primary registry and other registries are lost and have to be reestablished when the base registry adapts to its new position. When *base1* discovers that its primary registry reference is out-of-date it also attempts to locate a primary registry and finds the new primary registry *primary3*. If *primary2* had been within the latency bound for *base1* and *base2*, the base registries would have joined the second primary registry.

If, during the period that the primary registry is uncontactable, no base registries attempt communication there is no adaption required. This model allows adaption to occur as required by the system to minimise domain rearrangements. A registry can become aware of a missing primary registry through three cases: forwarding a registration, forwarding a deregistration, and forwarding a location query that it was unable to resolve. A registry continues to operate effectively on its own, irrespective of primary registry failure, until one of these events occurs.

A base registry that adapts to the role of a primary registry does not have access to the database of name bindings managed by the previous primary registry. There are two ways by which the new primary registry can retrieve this information. The first way is by contacting an administrative registry to see whether any cached information exists for that registry domain. The second way is for the primary registry to wait until it is contacted by base registries within its domain and to interrogate these registries for information about their registered services. This process can cause delays in object location and relocation.

An alternative mechanism for ascertaining connection validity relies on polling, where each base level registry intermittently sends data to the primary registry and checks for receipt. The polling method achieves registry system stability within a shorter time than the lazy method used in the DISCWorld system but incurs its own problems. If all base level registries realise that the primary registry has failed at the same time, then they all proceed with a broadcast check at the same time. This reduces the possibility of a new primary registry being located and increases the chance of multiple new primary registries being set up to cover the same registry domain.

The possibility of multiple base registries within the one registry domain adapting to primary registries is still possible in the lazy adaption model but is dependent on the load and activity within the system. For a system that performs frequent remote resolutions this situation is more likely than for a system with infrequent inter-domain communication. As more failures occur in this situation, the number of primary registries increases and the number of base level registries assigned to each registry domain decreases. This situation leads to greater levels of communication between primary registries and a less efficient system.

The model of adaption used within the DISCWorld ORB system supports a form of fault resilience. A registry domain is capable of surviving multiple primary registry failures with a tendency towards decreasing performance as registry domains shrink and more remote resolutions are required.

6.3 Mobility

Objects within a DISCWorld ORB system are able to migrate between ORB nodes by using the migration support services provided by the system. These services include the dynamic creation of mobile objects designed to minimise communication costs, and migration and relocation services for mobile service objects. Object migration of any kind within the DISCWorld ORB system occurs between specified access points for each ORB node. The communications models based on object migration are discussed further in Section 6.6.2.

Migration requires that the object be serialisable, *i.e.* it is able to be transported in a byte form that can be successfully interpreted and converted back into object form. The DISCWorld ORB system provides base mobile object classes and an API to assist in the creation of mobile objects. Objects that inherit from the provided mobile base classes can migrate to any node within the ORB system.

As described in Chapter 1, the DISCWorld ORB system classifies references into three categories: unknown, unconnected and connected. Unknown references are those held by clients that have yet to contact the ORB system and request resolution. These references consist of names and attribute sets that describe objects. Unknown references are always valid as they contain no location dependent information. Unconnected references are references that have been returned from a resolution request but have yet to be accessed by the client. Access is defined as a request to the remote object using the reference, such as a method invocation. Access sets up a connection between the reference and the remote server object. Connected references are those references that are actively connected, *i.e.* they have been resolved, returned and invocations are currently taking place using the reference. References may return from the connected state to the unconnected state through the lifecycle described in Section 5.4.3.

References within the DISCWorld ORB system are location transparent and partially independent. Location information may be contained within the reference but it is not essential to referential integrity. References are relocation transparent. A client object with a reference to a mobile object is unaware of any relocations as their reference maintains its validity throughout the migration.

Mobile objects within the DISCWorld ORB system support migration transparency. The registry system may decide to relocate an object without communicating this decision to the mobile object or to any of the mobile object's connected clients. The migration, including invocation queuing, updating and reconnection, is performed by the underlying infrastructure.

6.3.1 Location Independence and Location Transparency

References within DISCWorld are location transparent: they do not visibly contain or require the specification of a location. References consist of a name, which may be an alias, and a series of optional attributes. These attributes provide information about the referenced object such as a semantic description or platform requirements. When a name is resolved by the registry system, a reference is returned which contains cached information

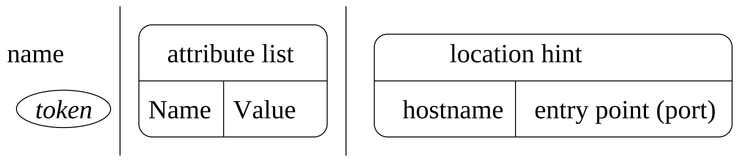


Figure 25: Internal structure of a DISCWorld ORB reference.

about the location of the object and any additional attributes specified by the client upon resolution request.

A DISCWorld reference is simply a name, implemented as a string value, with the addition of optional attribute information or location hints. A DISCWorld reference has the structure shown in Figure 25. This structure is divided into three sections, defined by the name definition, attribute specification and cached location hints. A connected reference contains a *token* which identifies any request queues contained within the server object's migration fragment. This identification is globally unique.

Attribute information contains tuples of attribute names and values. By storing the attribute information the client can review the specified attribute information; additionally, attribute information may be used to find a new service that matches the same criteria in case of server failure.

A location hint is held within the namespace database for each registered server object. This location hint corresponds to the location of the server object at the time of its last access, either at the object's registration or a resolution request for the object from a client. The location is provided as a hint only as the time delay between the capture of the location knowledge and the provision of that knowledge to a client may correspond to a migration period for the server object; hence, the information may be stale.

The inclusion of a location hint in a DISCWorld ORB reference means that these reference are not completely location independent. This location information is not required to maintain the validity of a reference but instead is used to increase efficiency by decreasing the need to contact the registry to correctly resolve the name for each communication.

6.3.2 Relocation Transparency

Relocation transparency in a mobile object system means that a reference to a mobile object is updated transparently when the object migrates. A relocation transparent reference is always valid regardless of the mobile object's location. DISCWorld references are relocation transparent as they maintain reference validity throughout object migration.

Unknown references are by definition location transparent as they can not contain any location information. An unknown reference is always relocation transparent as it can be resolved, regardless of the target object's location, by accessing the global namespace through the registry system.

Unconnected references contain a location hint that was current at the time of name resolution but is not necessarily current at the time of connection with the server. These references are relocation transparent as the unconnected reference can request an updated location from the registry system if the location hint is inaccurate.

Connected references contain a location hint that represents the location of the referenced object[9]. The location hint in a connected reference is explicitly updated when the referenced object migrates. To do this, the migrating object sends an update message to the client; this message is intercepted and interpreted by the client interface fragment. The client interface fragment can then reestablish the connection to the mobile object without communicating knowledge of the migration to the client.

Any invocation requests that overlap with the update request and migration time are queued by the migration fragment and reinvoked (in order) after migration. Upon reconnection, the client can immediately receive the results from its queued invocation requests.

Figure 26 shows the relocation mechanisms used in DISCWorld for connected references. Initially, (i) object A has a reference to object B. This reference is an unconnected reference and is location transparent. The unconnected reference contains a location hint provided by the registry system. If the location hint is correct, A's reference (in the form of a client interface fragment) attempts to connect to the specified location. This connection causes the creation of a migration fragment on B (ii). This fragment responds to requests and handles the connection to A only.

During connection establishment, a token is passed from the client interface fragment to the migration fragment. If this token is recognised by the migration fragment this indicates that a previous connection has existed between the two objects. In this case the same token is returned and any existing connection knowledge is matched to the new connection. If the token is not recognised then a new token is created that uniquely identifies the connection and is returned to the client interface fragment. This informs the client interface fragment that any previous connections that it had established have not been recognised. This failure may be due to server restart or failure.

In Figure 26 (iii), B is migrating. Migration of B causes its migration fragments to send location update messages to each of the clients managed by a migration fragment. These update messages are intercepted by the client interface fragment at each client and transparent reconnection is performed, as shown in (iv). Causality is maintained by processing requests in their receipt order; after migration, new requests can not be processed until all queued requests have completed. A transaction-based system could be implemented on top of these facilities to support recovery and rollback, however, these extensions are outside the scope of this thesis.

[9]The type of connection used by a client object to connect to the mobile object is dependent on the protocols supported by the client and the mobile object.

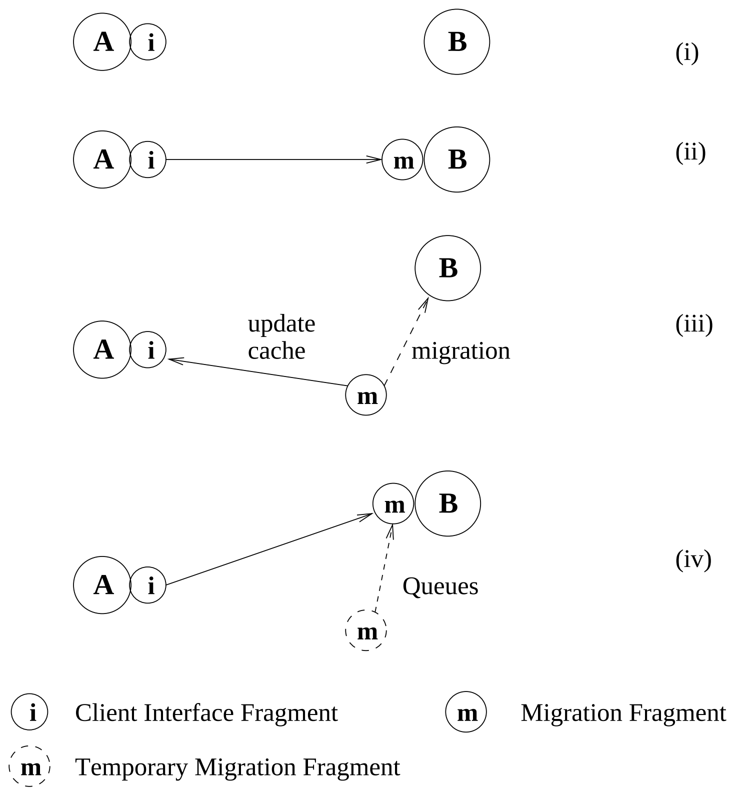


Figure 26: Location and relocation within the DISCWorld ORB update model.

The token contained within a connected reference is used to match up request queues after a migration and is also used to relocate the exact server object that the reference referred to in case of reference breakdown. This is done by explicitly requesting a reference update from the registry system.

Figure 27 shows the update mechanism in relation to registry updates for a migrating object. Figure 27 (i) shows the first stage of the update mechanism for a connected client in a registry domain system. In this example a client has a connected reference to object A on its local base ORB. When object A migrates, it forwards an update message to the client and deregisters from its local ORB. This deregistration message is forwarded to the primary registry of the domain.

Figure 27 (ii) shows the reconnection phase of the update mechanism. Object A has migrated successfully to another base registry within the registry domain and has registered with the local ORB. This register request is forwarded to the primary registry for the domain. After successful migration the client is able to reconnect its reference to object A using the cached location hint provided in the update message.

Resolution attempted during an object migration is resolved to a reference containing the location hint for the previously known location. Connections attempted using this location generally fail (unless the migration was rolled back) and cause the client interface

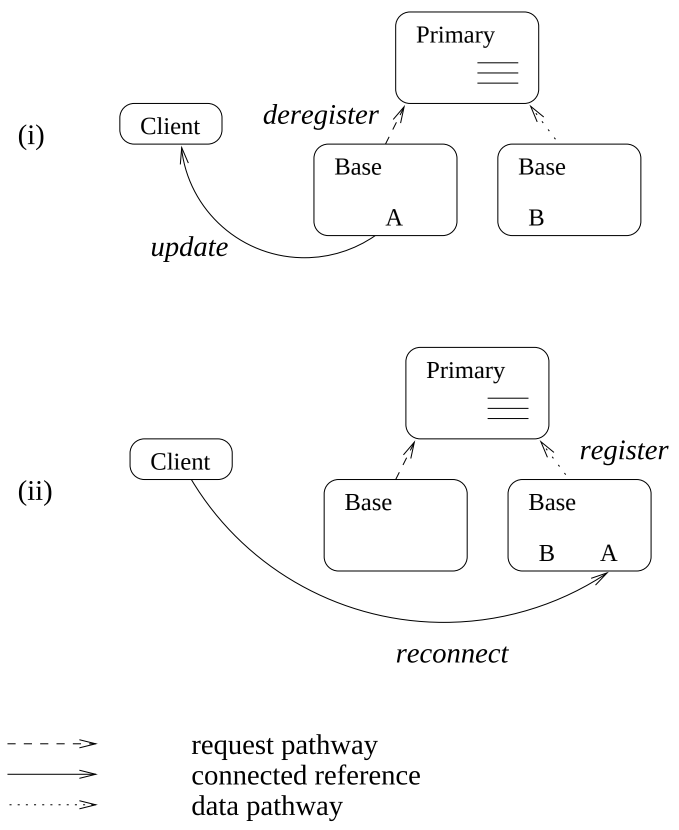


Figure 27: Distributed multiple registry relocation model.

fragment to contact the registry system again to find the new location. The benefit of this approach is that the client interface fragment is able to contact the registry node closest to the previously known location of the server object. The principle of locality suggests that a migration to a nearby node is more probable than a migration to a far node, hence it is probable that the server object is still within the same registry domain and can be relocated promptly. Optimisation of registry domain usage is most effective in a widely distributed object system with multiple registry domains.

Using the combined mechanisms of explicit updates and a distributed, hierarchically structured registry system provides relocation transparency for unknown, unconnected and connected references. Location independence and location transparency are also integral parts of this support with partial location dependence used to maximise connection efficiency while retaining the relocation validity of the reference.

6.3.3 Migration Transparency

The level of migration transparency within a system indicates how much knowledge about the migration is required or available to the migrating object. If the object is completely unaware of its migration the system is said to be completely migration transparent.

The DISCWorld ORB system supports a form of migration transparency where a mobile object can be migrated transparently in response to a request from the registry system. This mechanism provides support for the development of load balancing modules to be integrated into the ORB system. Mobile objects can also request their own migration, which is by definition not transparent. The object is necessarily aware of the migration but is unaware of the process of migration and any migration policies enforced by the system.

6.4 The Cost of Transparency

The costs involved in the operation of a distributed DISCWorld ORB system are divided into three categories:

- object location,
- service registration, and
- object relocation.

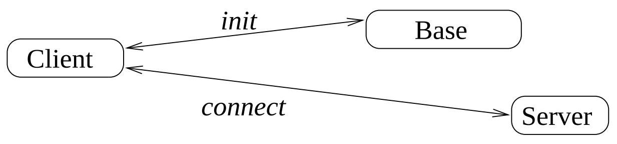
The costs for initial object location consist of the cost for a client to obtain a connected reference to a server, including the costs of registry contact, obtaining an unconnected reference, and connecting that reference to the service object.

The costs for service registration are the costs incurred by a server when registering itself with the ORB system, translating to the delay before the server object can commence satisfying client requests. These costs include communication costs between the server and the registry (dependent on the locality of the registry), any code loading costs and any policy-based costs (such as replica creation, cloning and dispersal costs).

The cost of object relocation includes the cost to a client when it must relocate a server object, and the cost to the server object of its actual migration time. The cost to the client consists of the communication costs for updates and any delays in request execution. The cost to the server is the time that the migration operation requires, including object transfer and initialisation on the new host. This cost is the maximum delay that any queued invocation requests will incur.

A base registry qualifies for inclusion into a registry domain if it is able to "ping"[10] the primary registry for that domain within a time n. If a primary registry can not be found within n, the base registry sets itself up as a primary registry for a new registry domain. Future base registries within the specified n boundary are then able to join the new registry domain. What can be inferred from this is that a poorly set n factor can lead to groupings of registries that are too large to be efficient (a large value of n) or to groupings

[10]A "ping" value is defined as the time taken to send a message to an object plus the time to receive the same message as returned by the object in a fashion similar to the UNIX command "ping" for performing latency tests between hosts.



init	Initial Registy contact cost
connect	Client-Server cost

Figure 28: Location cost for a local server.

that are too small (a small value of n) that cause large amounts of high-level inter registry communication.

The average latency distance between two base registries within the same registry domain is $l_{av} \leq 2n$.

The cost of object location depends on the latency between the client and its local base registry, and the location of the server object within the registry system. These location costs are described in Figures 28, 29 and 30.

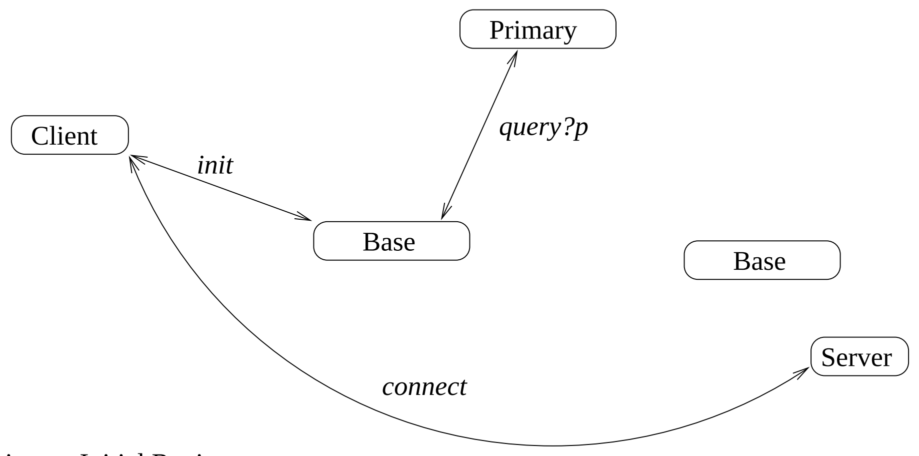
Figure 28 shows the cost for an object location where the server object is resident on the same host as the base registry contacted by the client. Information on the server object's bindings is contained within the database segment managed by the base registry. Given a cost *init* for a one-way communication with the base registry and a cost *connect* for a one-way communication with the server object, the client's cost for connection C_{init} is:

$$C_{init} = (2 \times init) + (2 \times connect)$$

where $init \approx connect$. Each communication consist of the assumption of a single packet, so a communication consisting of a single request across the *init* channel would be *init* while a request with a response would consist of $2 \times init$. The communication stages of *init* and *connect* can be separated by the client to increase concurrency as the *connect* associated cost can be delayed until the first remote invocation is required.

Figure 29 shows the cost for an object location where the server object is resident within a different base registry in the same registry domain as the base registry contacted by the client. Resolution has to proceed through the primary registry managing both base registries. Given a cost *init* for a one-way communication with the client's base registry, a cost $query?p$ for a one-way communication to the primary registry from the client's base registry and a cost *connect* for a one-way communication to the remote server object the cost for object location is:

$$C_{init} = (2 \times init) + (2 \times query?p) + (2 \times connect)$$



init	Initial Registy contact cost
query?	Update request to higher level registry
connect	Client-Server cost

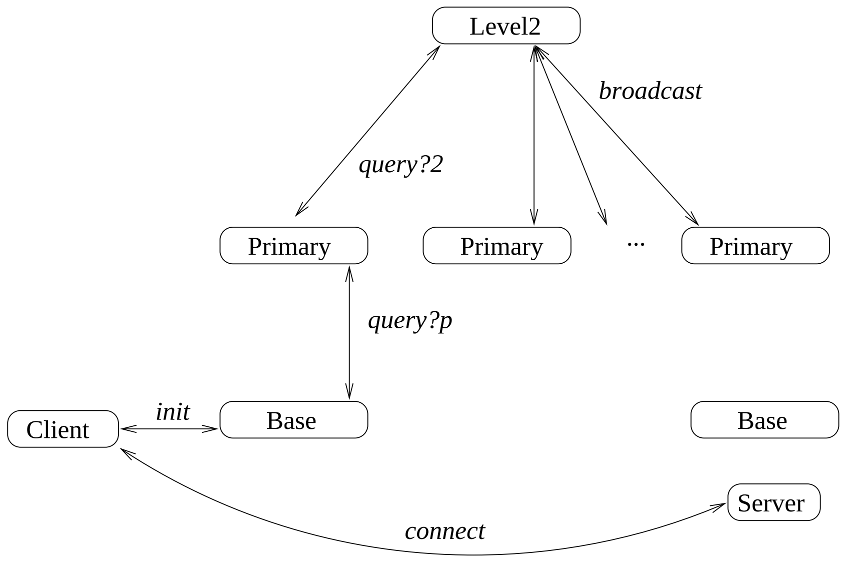
Figure 29: Location cost for a level 1 resident server.

where the cost of $query?p$ is guaranteed to be on average $query?p_{av} \leq n$. $query?p$ is the cost to query the primary registry.

Figure 30 shows the costs for object location where the server object is resident within a different registry domain to the base registry contacted by the client. Resolution has to proceed through the primary registry of the client's registry domain and into the administrative layer of the registry system. The client's primary registry must contact a level 2 registry and request resolution. The level 2 registry then broadcasts the request to all primary registries known to it. Assuming an initial one-way base registry communication cost of *init*, a one-way communication cost of $query?p$ to forward the request to the primary registry and a connection cost of *connect* to the remote server object the object location cost is:

$$C_{init} = (2 \times init) + (2 \times query?p) + (2 \times connect) + (\sum_{i=1}^{k-1} query?2_i)$$

The additional component of $\sum_{i=1}^{k-1} query?2_i$ corresponds to the cost of sequential broadcast of the query to each primary registry in the set of k primary registries known to the level 2 registry. Each broadcast has a cost of $query?2$. Each primary registry that receives a request replies only if its database contains the correct object. The level 2 registry waits for the first reply and forwards this result back to the client. If the level 2 registry does not receive a reply within a time limit of m it either forwards the request further up the



init	Initial Registry contact cost
query?	Update request to higher level registry
broadcast	Request update from lower level registries
connect	Client-Server cost

Figure 30: Location cost for a level 2 resident server.

registry hierarchy or, if there are no higher nodes, the requested resolution fails. This gives a minimum location cost of:

$$C_{init} = (2 \times init) + (2 \times query?p) + (2 \times connect) + (\sum_{i=1}^{k} query?2_i) + 1$$

and a maximum object location cost of:

$$C_{init} = (2 \times init) + (2 \times query?p) + (2 \times connect) + (\sum_{i=1}^{k} query?2_i) + m$$

Object relocation costs can be modelled by the cost of update, which corresponds to *connect* for each connected client interface fragment, and any migration delay. In the case of update failure and relocation for unconnected references the cost consists of the cost of a failed connection, $2 \times connect$, plus the corresponding registry-based location costs.

These cost models represent the full cost of object location. This cost takes into account the cost incurred by the client as well as additional costs of packet transmission over the network. The actual cost of object location using the DISCWorld ORB system consists of those costs directly effecting a client: these costs are the minimum costs of location as described in each category. As is examined further in Chapter 7, the costs of object location are affected by the additional packet transmission costs when network saturation levels are reached. As the number of clients or the number of client requests in a domain increases, the cost of location increases due to both the load on each ORB and the additional load on the network. It is also because of these reasons that it is desirable to obtain, through experimentation or dynamic adaption, the appropriate domain sizes for an ORB system.

6.5 Object Integration

Server and client objects can be dynamically integrated into the DISCWorld ORB system using the registration and resolution APIs provided by the system. These APIs allow server and client objects to specify details of their integration. For example, a server object can specify its mobility requirements, and a client is able to specify attributes that define the server to which it wishes to connect. These APIs are described in Appendix B.

6.5.1 Server Integration

Services can be integrated into the DISCWorld ORB system as either legacy or native objects using the registration APIs provided through library code packages. To integrate legacy objects a wrapping object must be implemented that is capable of registering the object's service and attribute set and is also capable of forwarding invocation requests to the legacy object. Native objects are able to register themselves.

The cost of registration consists of the cost of communicating registration information to a base registry. The selection of a base registry can be performed by the registry system API, in which case a search for a local registry is undertaken, or a registry location can be specified by the server object itself. The cost of forwarding the base registration to the primary registry is undertaken at the expense of the base registry and not the server object.

A service registration involves the specification of naming requirements, attribute sets, alias sets and policy requirements. The specification of these items is optional; registration can be performed by the registry system without policy specification. The specification of a name allows an object to request a specific binding. Depending on the naming model that is supported, (*i.e.* different conditions exist dependent on the model's support for multiplicity) this name may be rejected.

A registration may include the specification of attribute sets, where an attribute is a tuple specifying the attribute name and an attribute value. Attribute values can be of any type, including user defined types. The only requirement for a type to be an attribute value is that it must provide a mechanism for comparing two objects of the specified type for equality. Equality may be defined by the creator of the type.

Attributes may be specified as optional, mandatory, or guaranteed. This ability is required to allow the naming model to exhibit the registered and yellow pages preferences (see Section 5.1.2). In a naming system that supports the yellow pages preference a mandatory attribute is required in each attempted resolution. If the naming model does not have the required preferences, the attributes are interpreted as equally important, *i.e.* there is no preference ordering applied.

Aliases are defined as alternative names for the object. In registration, an object is able to define two alias sets that are to be used for it. These are defined as the local alias set and the global alias set. Depending on the naming model's support for aliasing, the local alias set defines aliases to be used within the same registry; the global alias set defines aliases that can be dispersed throughout the registry system.

A server object can define a policy that is used at registration to define what additional service the object wishes to use. These policies are used to control access to behaviour such as replication, cloning and mobility.

Policy Definition

A server object's policy defines how it is to be treated by the ORB system and also what additional services it is to have access to. A policy consists of a series of attributes that have been defined for that server object. A server object's policy also defines the alias sets for that object. The DISCWorld ORB system defines a basic policy object that allows definition and selection of the services provided by the system. To extend policy definition to enable the inclusion of services yet to be implemented, these policy objects have been designed to support extension and the selection of generic attributes.

The current base policy object allows specification of attributes used to control the replication, cloning and mobility services. The policy can also be used to define the communication protocols that the object supports. Policies can be defined that control the location of any created objects; clones or replicas can be distributed throughout the ORB system or can be isolated to one node. The ability to disperse objects also defines the object's ability to migrate.

Communications protocols can be chosen to reflect those protocols that the object is able to understand. These protocols define low level protocols, such as CORBA's IIOP.

The base policy defined by the DISCWorld ORB system allows these policies to be set through selection of attributes. The following attributes are provided for selection:

Replicate	Where
Mobile	Clients
Clone	Protocol

The *Replicate*, *Clone* and *Mobile* attributes are used to define truth values for their associated properties. For example, a value of *true* for the attribute *Mobile* is used to specify that the object is capable of migration. The attribute *Where* is used to define the distribution requirements for cloned and replicated objects, such as whether the created objects are to be distributed locally or globally.

The *Clients* attribute is provided so that a server object may restrict the number of clients that it will accept connections from at any one time. The *Protocol* attribute defines the number or type of communications protocols that are available. A policy that does have the *Protocol* attribute defined indicates that the server object can be contacted through any communications protocol supported by the system. To support a communications protocol, the server object must support the creation of fragment code that is able to understand the specified protocol. The details of this implementation are discussed further in Section 7.1.2.

As an example consider the case of a server object that wants to specify a policy for two object replicas to be created and distributed to different ORB nodes within the same registry domain. A policy that defines a request for replica or clone distribution must define the number of replicas or clones to be created, and must also define support for mobility. This requirement enables the registration system to verify that the object (and its replicas) is able to support migration. A sensible policy to define this has the following attribute settings:

$$\begin{array}{rcl} Replicate & : & 2 \\ Mobile & : & true \\ Where & : & domain \end{array}$$

An example server object registration for this policy is shown in Figure 31. This figure shows the creation of a server object and a policy object. The policy object is then set with the appropriate values for the required policy and linked with the server object using methods provided by the Registry API.

A server object is not required to specify a policy when registering. A default policy is used by the DISCWorld ORB system for objects that do not specify their own and can be set for each ORB within the system. The default policy used in a default DISCWorld ORB system supports object mobility within the registry domain. A policy to define these requirements has the following attribute specification:

$$\begin{array}{rcl} Mobile & : & true \\ Where & : & domain \end{array}$$

Object replicas and clones are provided with the same policy as their parent object with some alteration made by the ORB system. For example, a replica has the same policy as its

```
Server obj = new Server();
Policy policy = new Policy();
// Creation of new server and policy objects

policy.setReplicate(2);
policy.setMobile(true);
policy.setWhere("domain");
"
// Attribute setting

Registry.register(obj,policy);
```

Figure 31: Example server registration code.

parent without the requirement to replicate. The DISCWorld ORB system allows policies to be nested, enabling a server object to define separate policies for any secondary created objects. For example, an object requesting that clones be created may create a separate policy for its clones, or even for each of its clones. These policies can then be used to define further replication or mobility and protocol selections.

Policies are interpreted and implemented by the base registries within the system. Policies are maintained as part of the object information within the database and are forwarded with the object upon migration. Policies are not strictly re-interpreted after the first interpretation; for example, if an object and its replicas are migrated to another host, a new set of replicas is not created.

A server object is able to change its policy during its lifetime using APIs defined by the DISCWorld ORB system. This allows the server object to expand its replica set, to change its migration policy temporarily, or to alter the number of acceptable clients in response to load or demand changes.

More information about policy specification and the base policy provided by the DISCWorld ORB system is provided in Appendix C.

6.5.2 Client Integration

Client objects can integrate and communicate with objects within the DISCWorld ORB system using the resolution and object location mechanisms provided by APIs within the system. The APIs provided by the system allow two main forms of client integration with the ORB system. The first form is through name resolution, where a client already knows a name and wishes to resolve that name to a reference that can be used to communicate with the server object. The second form is through attribute-based resolution. Attribute-based resolution matches a set of attributes defined by the client against those provided by registered server objects according to the currently specified naming model.

```
ServerInterface obj;

obj = (ServerInterface)Registry.lookup("Webserver");
// Resolution using explicit name.

obj = (ServerInterface)Registry.lookup("moonstone", "Webserver");
// Resolution using explicit name for specified ORB.
```

Figure 32: Example name resolution code for a client.

```
AttributeList attributes = new AttributeList();
attributes.ordered(true);

attributes.add("host_type", "alphastation");
attributes.add("service", "webserver");
```

Figure 33: Example attribute set specification.

To resolve a known name to a server reference the client must specify the name in a request to the ORB system. An example of this specification is shown in Figure 32. This example shows two name resolutions for the name *Webserver*, both using the Registry API. The first resolution requires the registry system to locate a local ORB node through which to make the request. The second resolution includes an explicit request for resolution to be performed through an ORB node located at a host specified by name *moonstone*.

Attribute-Based Resolution

Resolution can also be performed through the specification of attributes that define an object. This specification may be narrow and in the form of the unique name of an object or it may be wide and define a set of semantic attributes that an object should or must support.

Attribute specification is performed by creating a list of attributes that are required in the matching object. As specified in Section 6.5.1, attributes can be of any type as specified by either the client or server. Attributes can be used to specify a service type, cost requirements, a host type or a locality requirement. Figure 33 shows an example of attribute-based resolution using the DISCWorld ORB system. This example shows the specification of an attribute set for ordered attributes. Attributes that define the required host type, *alphastation*, and a specified service type, *webserver*, are added in their defined order to the attribute set.

An example of attribute-based resolution using the DISCWorld ORB system's resolution model is shown in Figure 34. This example shows resolution requested on a local ORB node for the attribute set as specified in Figure 33. This example shows two resolution attempts.

```
Object server;
server = Registry.locate(attributes);

Object[] servers;
servers = Registry.locateAll(attributes);
```

Figure 34: Example attribute-based resolution.

The first resolution is a requested resolution for the first matching object. The second resolution is a requested resolution for all objects that match the attribute specification.

Intuitively, a request to resolve all objects that match an attribute specification requires a complete database examination. In practice a client may require multiple matching objects but may not require that each matching object within the entire ORB system be returned. Using the APIs provided by the system, a client can specify the level to which they wish their resolutions to proceed. For example, a client can specify the level *domain* as the highest level to proceed to in a successful resolution. A client can also specify an alternative level to proceed to in the event of an unsuccessful resolution. The difference between these values is that in a successful resolution a subset of the matching objects within the system has already been determined. An unsuccessful resolution means that an empty set of matching objects is currently the determined set.

The naming model that defines the ordering of requests as required by a client can be defined by each client. For example, if a client wants their resolution results to be ordered according to the registered preference they are able to specify a sub-naming model that is applied to their results only.

An attribute-based resolution system is only useful when the attributes specified by the client is a subset of those registered by a server. A client must either be programmed with the knowledge required to use the attribute system or there must be a mechanism provided to enable the object to learn about the attribute system. To gain knowledge about the different attributes specified by server objects within a DISCWorld ORB system, a client can request a listing of those objects currently registered with an ORB. The information returned in this listing contains the attributes that describe the object. These attributes can be used to *bootstrap* a client into an ORB system by enabling it to specify attribute-based resolution requests with full knowledge of the attributes used within the naming system.

6.6 Communication Models

The DISCWorld ORB system supports two models of communication. The first model is an RPC-style [20, 22] model involving a fragmented object model. In this model, the client interface fragment acts as a client-side proxy for the server object. The proxy is able to transparently invoke methods, both synchronously and asynchronously, on the remote server object. The RPC-style model is capable of using multiple low-level protocols including the native DISCWorld ORB protocol, Java's JRMP [70] and IIOP [136].

The second communication model is a mobile communication model allowing the dynamic construction of programmed agents. To maximise communications efficiency, a client can construct a mobile communication agent that is able to migrate to the hosts where services are present and invoke those services locally. This model of communication minimises the number of remote communications that must be performed. These agents are semi-autonomous in that they can vary their programmed itinerary in response to external changes in the ORB system. Mobile communication agents can be created using APIs provided by the DISCWorld ORB system. Mobile communication agents are capable of acting as DISCWorld ORB services in their own right.

6.6.1 RPC-style Communication

The RPC-style communications model used in the DISCWorld ORB system [52] is similar to that used in fragmented object systems such as Hobbes [121,163] and middleware systems such as CORBA [18, 136, 138] and Java RMI [167]. The fragmented object model used to maintain transparency within the ORB system is also used to provide communications, with the client interface fragment acting as an invocation proxy. The client interface fragment is responsible for translating the invocation request into a transportable form and sending the request to a matching migration fragment. The migration fragment is responsible for interpreting invocation requests and invoking the requested method on the server object. The migration fragment must also send any result from the invocation back to the client interface fragment. The client interface fragment has the same method signature as the server object and allows invocations to be invoked on the proxy as if it were the server object.

This form of communication is very similar to that used in many distributed object systems. Many of these systems support only a few variations on the basic communications model. For example, most systems support synchronous method invocation only. It is possible to create specialised implementations of these communications protocols, but generally these implementations are not compliant with the standard synchronous implementation. This leads to a reluctance on the part of the software developers to explore optimised or extended protocols.

Java's RMI mechanism is an example of a standardised communications protocol that provides a limited set of optimisations. All method invocation is performed in a synchronous manner unless explicitly subverted by the client and server implementations. Implementations of CORBA over Java allow for one-way or asynchronous method

```
public class Client {
          Server FutureServer;
          // Server is bound to Remote Server.

          Future ResultValue = FutureServer.Method1();
          // Client continues with execution.

          ResultValue.complete();
          // Client blocks until Remote Server returns a value.
 }
```

Figure 35: Example code fragment utilising future objects.

invocations using IIOP as the underlying protocol. These costs have been investigated in [66, 67].

The DISCWorld ORB system supports multiple communication models: namely synchronous invocation, asynchronous invocation and delayed referencing of return objects (Futures [183]). A server object within the DISCWorld ORB system can define, through policies or through static code specification, which of its methods (on a per-method basis) may be invoked in a synchronous or asynchronous manner and, if a remote object is non-void, whether its return object is a future object.

Delayed referencing of future objects allows the client to obtain a reference to the return object of a method invocation as soon as the asynchronous method returns, but delays the retrieval of the result until the client explicitly accesses the value. This may result in an optimisation in two ways: calculation of the result value can continue concurrently with client execution until the value is requested; and if the client is not interested in the return value of the method, but only in causing the method invocation itself, the future value may never need to be retrieved. Future objects as defined in [183] and are similar to models used in Promises [117] and DRAMFs [81].

Future objects delay any blocking on remote method invocation until the result is required by the client. This is especially useful when a client is not interested in the result of a method invocation, but only in the invocation occurring, avoiding unnecessary object transferral. An example code fragment that utilises a future object as implemented in the DISCWorld ORB system is shown in Figure 35. The local client and remote server can both continue execution until the client explicitly requests that the future object be returned. Alternatively, greater optimisation can be achieved by batching delayed references. By bundling several requests together the communications overhead required is minimised for the invocation and result retrieval. This is known as Batched Futures [24]. The dependencies required to utilise systems such as Futures and Batched Futures have been studied in [45].

This communications model allows the specification of variations on the traditional RPC-style communications model through the use of client interface fragments. All of these forms share a common cost: the cost of parameter and result object transmission. In the case of multiple method invocations, where the result of one invocation is passed as a parameter to the next invocation, the cost of these transmissions can become large. For this reason the RPC-style communications model is best suited for low latency remote invocations where each invocation is separated in its effect from the others.

6.6.2 Mobile Communication

As described in Section 6.6.1 traditional remote invocation is not suited to the cases where the client and server have a high latency in communication and the required method invocations are highly connected. In these cases it is best to move the invocation to the remote sites rather than transport all of the required parameters and results multiple times. This form of communications is a mobile agent-based model.

The DISCWorld ORB system introduces a communications model based on a dynamically created agent-based communications object. These objects can be programmed with *itineraries* and *tasks*. Itineraries correspond to services or objects that the agent is required to visit; tasks correspond to methods that are to be invoked on the objects once a reference is obtained. This thesis introduces the term *mobile communicator* for this kind of communications object.

Mobile communicators are mobile objects designed to take advantage of the ORB model of communication. Constructed in a similar way to a mobile agent, a mobile communicator contains all the information it requires to migrate around a registry system. A mobile communicator can be programmed to resolve names to objects, then migrate to the location of the referenced object, a *voyage*, to perform location invocation, a *task*. The results of one invocation can then be used in future invocations without unneccessary object transmission.

Mobile communicators are not programmed with the locations of objects in order to preserve the location and relocation transparency inherent in the DISCWorld ORB system. Instead, mobile communicators are given information required for object resolution, such as a name or a defining attribute set. This information can be used to resolve a reference using the underlying ORB system when the reference is required. The mobile communicator can then request to be colocated with the referenced object allowing local invocation.

Figure 36 shows an example of a client using a mobile communicator to invoke methods on remote server objects. In this example a client has created a mobile communicator to invoke two methods on an object identified by the name *webserver*. The first method to be invoked, *giveName*, does not return a value of interest to the client. The second method to be invoked, *listBackEndServers*, returns an object array that represents a list of known back-end server objects. This result is required by the client and is hence recorded within

```
MobileCommunicator mobj = new MobileCommunicator("webserver");

mobj.addVoyage("webserver");
mobj.addMethod("giveName");
mobj.addMethod("listBackEndServers",new Object[],"varA");

mobj.start();
```

Figure 36: Itinerary for a mobile communicator.

```
Mobile mobj = (Mobile)Registry.lookup("mobileCommunicator_webserver");

if (mobj.Completed()) mobj.start();
```

Figure 37: Monitoring code for a mobile communicator.

the mobile communicator under the name *varA*. This name can be used to pass the result from this second invocation to future invocations.

Mobile communicators exist as ORB services in their own right and can be contacted through name resolution by other objects within the registry system. In this way, a mobile communicator can be remotely monitored and have its actions either restarted or cancelled. A mobile communicator can be programmed to permanently reside at a remote location with intermittent result retrieval performed by the initiating object.

A mobile communicator can be given a name that can be used to locate the object anywhere within the DISCWorld ORB system. This reference can be used to monitor and interact with the mobile communication agent. Figure 37 shows an example of the monitoring of a mobile communicator using the interfaces provided by the DISCWorld ORB system. In this example a client is monitoring the mobile communicator created in Figure 36 and restarts the object when it has successfully completed its tasks. The client obtains a reference to the mobile communicator regardless of its current location using the name bound to the communicator at its creation.

This mechanism is intended for widely distributed ORB systems with applications in repetitive tasks, system monitoring, logging and information gathering. The main advantage of this construct is that it only needs to be sent from the client once, regardless of how many invocations are required. Hence, a structured repetitive service can be established and invoked repeatedly.

6.7 Summary

The DISCWorld ORB system is a distributed object system that supports transparent location and relocation of named objects. The ORB system is designed to support transparent client/server communication within the DISCWorld metacomputing environment, as well as providing additional support for scheduling tasks and object management such as migration, replication and cloning.

The DISCWorld ORB system utilises a generic naming model where the required naming model for an ORB node can be specified dynamically using the formal naming model designed and expanded upon in Chapters 3 and 5. Each ORB within the DISCWorld ORB system manages part of a global namespace distributed throughout the hierarchically structured distributed ORB system.

Transparent location and relocation of objects is managed using an explicit update model and the distributed namespace as a default location mechanism. A fragmented object model is used to provide suitable mechanisms for the development of communications models.

Server objects can be integrated into the ORB system through the use of APIs provided by the system. Server objects can define policies, attribute specifications and aliases using the provided APIs. These polices control how the server objects are managed by the ORB system. Clients can access objects throughout the ORB system using resolution APIs that provide for explicit name resolution and attribute-based resolution with support for the name resolution models defined in Chapter 5.

Communications models have been designed to support traditional RPC-style communication and a unique mobile communication form. RPC-style communication can be performed synchronously, asynchronously and using future objects. Mobile communicators allow semi-autonomous agents to be programmed with location independent itineraries. Mobile communicators can be used to minimise parameter transfer costs in connected remote invocations. These objects are long-lived and can be used to set up repetitive services that can be remotely monitored.

Chapter 7

Implementation

The implementation of the DISCWorld ORB system consists of a prototype ORB and distributed ORB system that supports the naming, transparency and communication models developed in this thesis. This implementation includes an adaptive ORB system, a distribution mechanism to manage the global namespace throughout the hierarchically structured ORB system, and the implementation of the models themselves.

To facilitate development and testing of the DISCWorld ORB system on multiple hardware platforms, the system is implemented in Java [69]. Java is a platform independent language that has support for object cloning, serialisation and reflection. The choice of Java as an implementation language has enabled the development of the DISCWorld ORB system and additional services to take advantage of the language's support for object manipulation.

7.1 Implementation of the DISCWorld ORB

ORBs within the DISCWorld ORB system are capable of taking a position at any level within the distributed hierarchy. Repositioning is performed in response to an initialisation request or to a node failure. To implement this adaptive nature, each ORB must be able to assess its requirements and the requirements of its environment during the ORB's creation and also progressively during the system's operation. Each ORB provides APIs that describe ORB-provided services available to clients, servers and other ORB nodes within the system. This interface acts as an abstraction to the connection details and the underlying communication protocol used throughout the system. The content of each ORB's API is dependent on which services it supports and, also, its position within the node hierarchy. The APIs available in the DISCWorld ORB system are defined in Appendix B.

Each ORB consists of a collection of control threads and a database segment. Each control thread manages a service for that ORB and provides a contact point for external objects such as clients, servers and other ORB nodes, using the interfaces provided by the ORB. Each thread has access to the database segment managed by that ORB through a standard interface. Service management threads can be defined for each service that is required within the ORB system.

The service management threads provided by each ORB include a mobility thread and a query thread. The mobility thread provides a contact point that migrating objects can use as a target destination. This thread initialises newly migrated objects at their new location and performs registration on their behalf within the database. The query thread acts as a contact point for queries and requests related to the naming system, including name binding, management and resolution. An additional thread is created by primary registries. The primary thread manages a contact point through which base registries within its registry domain can contact it.

Figure 38 shows the architecture of a level 1 DISCWorld ORB. This figure shows the interfaces, objects and threads that make up an operating ORB. Each ORB is accessed through a common interface: the `Registry` interface. The threads managed by each ORB consist of the `QueryThread`, `MobileThread` and an optional `PrimaryThread`. `QueryThreads` and `PrimaryThreads` manage database segments that are accessed through the `RegistryData` interface. A primary ORB creates all three threads and maintains two database managers: one to manage the segment belonging to the registry and one to manage the segment belonging to the registry domain. A base ORB creates only the `QueryThread` and the `MobileThread`.

Data stored within a database segment consists of the set of names, the set of objects and the set of *(name, object)* bindings known within that segment. This information is stored using a two-level process. All `Names` known to that segment are stored in a lookup table; this table can be used to find out whether the name has been bound to any objects during its lifetime. The information contained within the lookup table includes a `Key` object. A `Key` object is used as an index into a hashtable that contains all of the bindings known to that segment. Once a `Name` has been located in the lookup table, the name's `Key` can be used to locate the list of objects bound to that name. This list can then be searched further to locate an appropriate object.

Level 2 or administrative ORBs maintain a cache of the locations of registry domains and information returned from past queries. This information can be used to help rebuild primary registry database segment information in the case of failure and required adaption.

Each service management thread creates a port for listening to incoming requests from external objects. Requests are defined in terms of the DISCWorld ORB communications protocol. This protocol defines an extensible framework for describing requests and tasks upon a database segment using the standard interface provided for a database manager. Database manager objects are described in more detail in Section 7.1.1, while the communications protocol is defined in Section 7.1.2 and Appendix A.

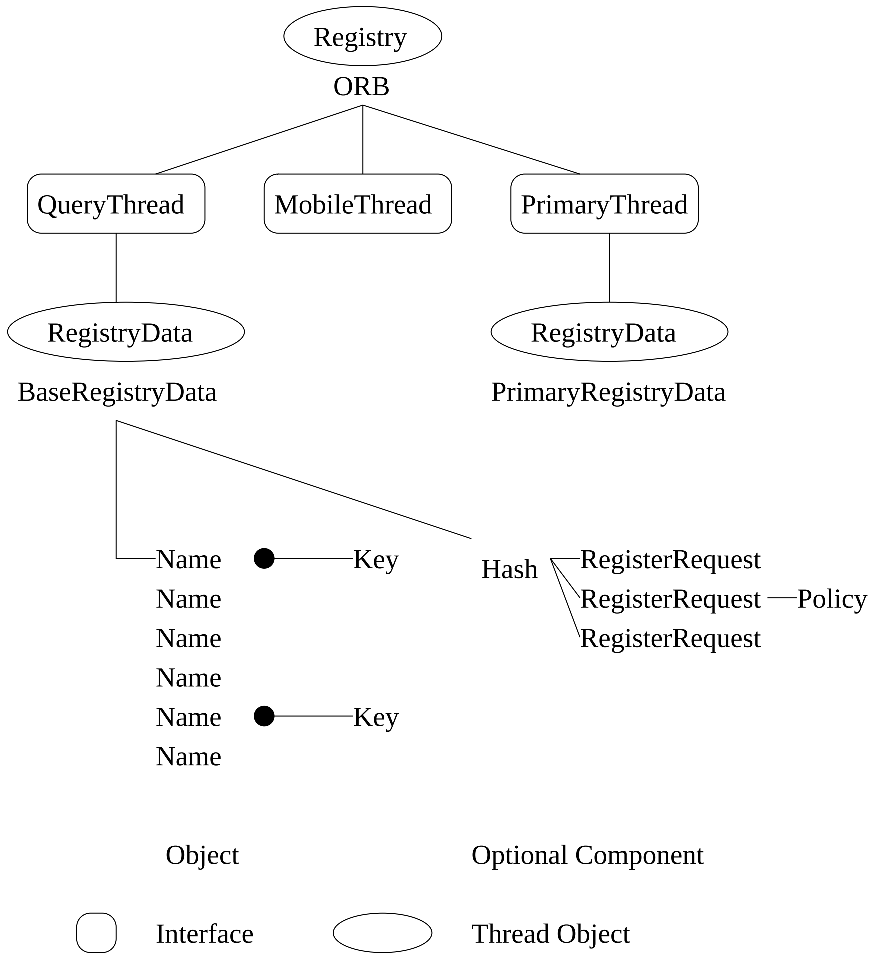


Figure 38: Architecture of a level 1 DISCWorld ORB.

7.1.1 Database Manager

Each ORB manages a segment of the global database or namespace. To correctly manage a segment, an ORB creates a database manager of the appropriate level for the ORB's position within the distributed ORB hierarchy. Each database manager supports the standard interface regardless of its level. Database management objects have the ability to adapt to manage segments at different levels as required by the system.

The standard database interface presents an interface through which common database operations can be performed. These database operations are performed in response to requests from external client and server objects through the use of the provided ORB APIs. By providing a standard interface, no external object can directly access the database. Changes to the database can only be performed through the operations deemed allowable by the database manager.

In general the database manager provides interface operations that correspond to adding information to the database (binding), removing information from the database (unbinding), changing information in the database (rebinding or managing) and querying the contents of the database (resolution). Each of these operations executes within the constraints of the current naming model of the ORB. To verify that the request is allowable under the current naming model, these functions make use of implementations of the complete binding, management and resolution functions as defined in Chapters 3 and 5.

Each database segment consists of the set of names, objects and bindings known to the ORB managing the segment. Database segments are implemented as hashtables, indexed by a key that is linked to each known name. Each alias for a name is given the same key. A complete set of names known to the ORB is separately maintained in alphabetical order in order to provide additional mechanisms to obtain key information. To allow the database to support multiplicity, each entry in the binding hashtable corresponds to a queue of objects that are bound to the name. This mechanism for managing the database is a two-level resolution mechanism designed to natively support aliasing and multiplicity. The more restrictive forms of these attributes can be implemented within these bounds.

Figure 39 shows the implementation of the bind function for a database manager used to manage database segments for primary level ORBs. In order to add a binding to the database several model verifications and resolution steps must be performed. Initially the policy assigned to the new binding must be verified; this process adds the name to the set of known names if the naming model supports a dynamic domain. The set of names is then searched to find the corresponding database hashkey. The database is then searched to access information about any objects already bound to the name. If the complete binding function validates the binding then the binding is added to the database.

Some operations provided by the database managers require that segments outside of those managed by this ORB be examined. For example, a request to add a binding to a database within a system that supports name reuse will only complete the binding if the object is no longer known within the system. This verification requires that the entire database be searched. To perform these forms of requests an ORB forwards the request to higher levels within the system. To perform a complete check on the database, the topmost ORB node must verify that all database segments are compliant with the requirements for the request. In the example of name reuse, the topmost ORB must verify that the object does not exist within any of the database segments.

Information is maintained within a registry domain so that each primary registry has a consistent view of the database segments that it is managing. To communicate bindings, or verify requests, base-level ORBs forward their requests to the primary ORB for their registry domain. Each primary ORB is then able to provide consistent database information to administrative nodes.

```
public void bind(RegisterRequest service) throws PolicyException,
                                                BindingModelException {
    Name theName = policyCheck(service);
    // Checks policy and adds name to name set if model is dynamic domain.

    Key index = theName.getKey();
    // Each Name has a unique key into the database.

    Queue element = null;

    if (database.containsKey(index))
        element = (Queue)data.get(index);
    // Gather information if the name is known.

    if (!completeBindingFunction(element,service))
        throw new BindingModelException();
    // Check naming model for correctness of binding.

    if (element == null)
        data.put(index,new Queue(index.toString()));
    element = (Queue)data.get(index);
    // Add a database object to manage this name if unknown.

    element.add(service);
    // Add binding.

    primaryRegistry.add(service);
    // Communicate binding to higher ORB levels.
}
```

Figure 39: Implementation of the database manager `bind` function.

7.1.2 DISCWorld ORB Communications Protocol

Clients, servers and ORB nodes communicate using the DISCWorld ORB communications protocol. This protocol defines the operations and communications that are allowed between objects within the system. The APIs provided by the ORB system provide an abstraction to the communications and protocol operations that occur between the client interface fragments, migration fragments and ORB nodes.

The DISCWorld ORB communications protocol consists of the transmission of defined protocol objects between the components of the system: clients, servers and ORB nodes. Some of the protocol object transmissions are synchronous and some are asynchronous, depending on whether failure notification is important for the specific type of request. Protocol objects are self-evaluating and do not need to be externally interpreted.

The DISCWorld ORB system provides a set of protocol objects to support the common operations and requests between clients, server and ORB nodes. The set of protocol objects

```
public synchronized static void register(base.RemoteObject service,
                                          registry.protocol.Policy policy) {
    connect();
    // Connect to a local ORB.

    adapter = new AdapterThread(service);
    ((base.RemoteObject)service).setAdapter(adapter);
    // Create a migration fragment adapter for the new service.

    RegisterRequest obj = new RegisterRequest(service,adapter.getPort());
    obj.setHost(adapter.getHost());
    obj.setPolicy(policy);
    send(obj);
    // Create a request protocol object to register the new service.

    ResponseObject response = receive();
    // Wait for response from ORB node.

    if (!response.getAchieve())
        throw new RegisterException(response.getAchieveData());
    // If registration was unsuccessful then raise an exception.

    adapter.start();
    disconnect();
}
```

Figure 40: Registering a server object using the DISCWorld ORB APIs.

available to each type of external object is restricted; for example, a client object is only able to receive and send a subset of the protocol objects that are available within the system. These subsets are controlled by the abstractions provided by the interface fragments and system APIs. Protocol objects contain all of the information that is required to perform the requested operation on the target object. Specific protocol objects are designed for each type of request supported by the system APIs. For example, there is a specific protocol object designed to support service registration.

Figure 40 shows an example of usage of a protocol object designed to register a service with a local ORB. This example is part of the `Registry` API available to client and server objects and is the method used to integrate a server object shown in Figure 31 (see Page 163). This method is responsible for registering the service with the ORB system and manipulating the server object so that it is able to receive client requests.

To enable a server object to receive requests, the `Registry` API must create a migration fragment adapter. This adapter is responsible for receiving queries from the ORB system and for creating migration fragments for connected clients. After the server object is successfully adapted, a `RegisterRequest` protocol object is created and sent to a local ORB.

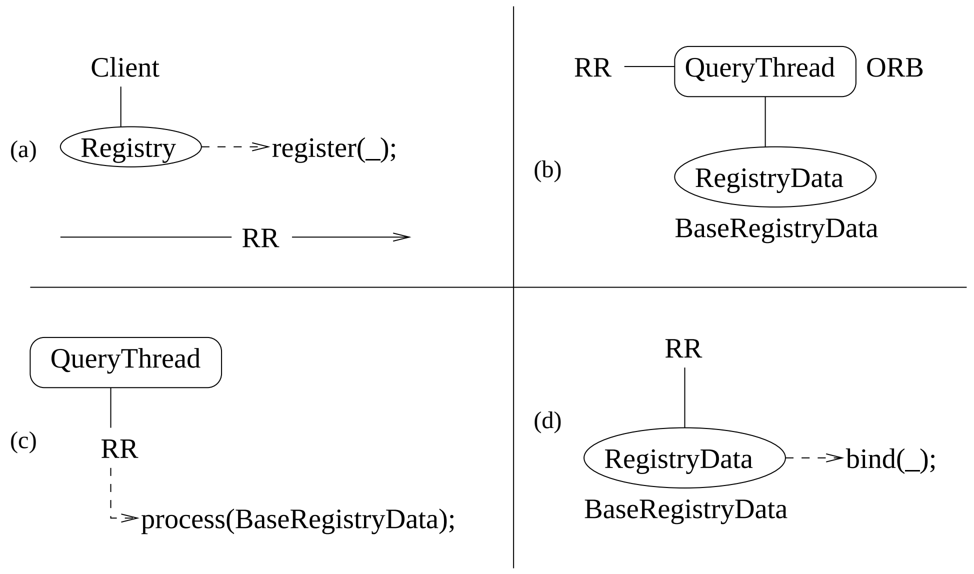


Figure 41: The process of activating a protocol object.

In this example a `ResponseObject` protocol object is sent back to the server from the ORB. A `ResponseObject` contains information describing the success or failure of the requested database operation. If, for example, the binding conflicted with the ORB's current naming model, the `ResponseObject` describes this failure.

A framework has been developed to support the future creation of protocol objects. Each protocol object supports a standard interface that enables it to record commonly required information and to apply its requested operations to a database segment managed by an ORB node. Each protocol object extends a base protocol object `ProcessRequestObject`. This object stores information about the client, server and ORB objects involved in the communication and the target name reference, whether it be for resolution, registration or management. The `ProcessRequestObject` also contains a process method that is responsible for invoking any database operations that are required. Each protocol object is required to implement this method.

When a protocol object is received at an ORB node, the service management thread that received the object is responsible for interpreting the protocol. The service management thread calls the process method, passing it a reference to the database manager for the current database segment. The process method can then request operations on the database segment using the database manager interface. Figure 41 shows the process of the completion of a register request using the DISCWorld ORB communications protocol.

Figure 41 (a), shows a client requesting a registration through the use of the `Registry` interface. This request results in the creation of a `RegisterRequest`, indicated by *RR*, that is then sent to a local ORB. In (b) the `RegisterRequest` arrives at the ORB where it is intercepted by the ORB's `QueryThread`. In (c) the `QueryThread` processes the protocol

```
public void process(RegistryData ds, ObjectOutputStream out)
                    throws ResponseCorruptionException {
    ResponseObject response = new ResponseObject();
    // Create a Response Object to return to the client.

    try {
        ds.bind(this);
        // Invoke operation on database segment.

        response.setAchieve();
    }
    catch (Exception e) {
          response.setAchieve(e);
    }
    try {
        out.writeObject(response);
    }
    catch (IOException e) {
           throw new ResponseCorruptionException(e);
    }
}
```

Figure 42: Example process method for the `RegisterRequest` protocol object.

object by invoking its `process` method and passing it a reference to the database segment managed by the ORB. In (d) the `RegisterRequest` object invokes its required operations on the database segment through the `RegistryData` interface. The `RegisterRequest` requests an addition to the database by invoking the `bind` method.

Figure 42 gives an example of the process method for a `RegisterRequest`. In this example, the process method is performing a synchronous protocol communication. The process method must create and return a `ResponseObject` that records the success or failure of the requested registration as well as performing the registration through a call to the `bind` method of the appropriate database manager.

Appendix A gives an extended example of the implementation of the `RegisterRequest` protocol object and provides the framework upon which protocol objects can be constructed. The set of protocol objects currently used within the DISCWorld ORB system is also described in Appendix A. Through the restricted APIs provided by the DISCWorld ORB system, each type of external object (client, server or ORB node) has a restricted set of protocol objects that it is able to send and receive. The interactions between the classes of external components and the protocol objects are modelled in Appendix A using a form of Lynch and Tuttle's I/O Automata [118].

7.1.3 Distribution Mechanisms

The distributed global namespace is managed by a distributed, hierarchically structured ORB system consisting of two levels: active and administrative. ORBs within the active layer, base and primary ORBs, are responsible for maintaining database segments for their part of the global namespace. A binding is the responsibility of an ORB if it is registered with that ORB or is within its registry domain. Administrative ORBs do not maintain a consistent database segment for their set of registry domains. Instead, administrative ORBs cache segments and perform on-demand verification of segment information. This distribution mechanism is a mixture of the push and pull models of information flow as discussed in Section 6.2.

Clients access remote server objects through the distributed resolution mechanisms provided by the system APIs. These mechanisms support a node preference as described in Chapter 5. Through these mechanisms clients can remotely communicate with server objects using RPC-style communication or by using mobile communicators.

Server objects are distributed throughout the ORB system through migration, replication or cloning. Migration is semi-autonomous; mobile objects can request migration to a specific host or can migrate through colocation with a remote object. Mobile objects can also be relocated in response to external requests. The implementation of migration is discussed in more detail in Section 7.3. Replication and cloning of server objects are controlled by two service management threads. When a service is registered with an ORB node, its policy is verified and executed. Requests for replication and cloning are then passed on to the appropriate service management thread.

The process of replication involves creating the specified number of replicas of the server object and linking these through replication multiplicity to the name of the server object. Future access through this name references all replicas. The cloning process creates the specified number of clones for the server object and links these together through service group multiplicity to a single name. Future access through this name references a single clone. Selection of a single clone is performed in accordance to the naming model preferences for locality or node.

Replicas and clones can be distributed throughout the ORB system with the restriction of domain distribution or global distribution as specified in the policy for the server object. Objects to be distributed are first distributed throughout the domain as one object per node. If there are still objects yet to be distributed once all domain nodes are used then the objects are further distributed in one of two ways. If the policy for distribution is restricted to a domain distribution then the objects will be repeatedly distributed throughout the domain to give an even distribution. If there is no restriction on the object distribution then the objects are distributed to nodes outside the registry domain.

7.2 Naming System

The DISCWorld ORB system supports a generic naming model based on the extension and formalisation of the naming models defined by Bayerdorffer [15,16] and Bowman *et al* [26]. The generic model relies on the implementation of binding, management and resolution functions for each attribute or preference as defined in the extended naming model (see Chapter 5). These functions verify that the request made to them, either a binding or a resolution, is valid with respect to their part of the model. For example a management function for the single binding attribute will verify that the database does not contain any bindings for the requested name. To apply these functions to the global namespace database, an implementation of the complete binding, complete management and complete resolution functions must be performed.

The naming model for an ORB or an ORB domain is defined through the specification of attributes and preferences that are to be supported (as defined in Section 5.4.1). The selection of an attribute is done by including the attribute's name within the file or object containing the current naming model specification. The name specified in the attribute's inclusion must correspond to the name of the method implementing the formal definition of the attribute's application.

Each ORB is able to apply its naming model to the database segment that it is managing. Some of the binding functions require that the entire database be examined, hence these functions must proceed to contact nodes at higher levels and examine remote database segments.

7.2.1 Example Naming Model Functions

The implementation of the formal functions defined in Chapters 3 and 5 requires self-contained functions that verify their part of the naming model for the binding, management or resolution information given to them. The implementation functions corresponding to the formal definitions for each attribute and preference are provided in three model classes; each category of the naming model is implemented in its own model class. These model classes can be extended to support further extension to the naming model.

Each of the naming model functions within a naming category must have the same method signature to enable inclusion in the generic naming model. The required method signature for binding and management functions is defined as follows:

```
public static Boolean <functionName> (Queue thisQueue,
                                      RegisterRequest thisOne);
```

The method name given to the implementation of the formal function definition can be derived from the name of the attribute. For example, the method to implement the single binding attribute has the name singleBinding. This name must correspond to that included in the specification of a naming model. Each binding function is provided a subset of the ORB's database segment that is the set of objects bound to the name specified in the `RegisterRequest` protocol object. This subset of the database is provided as an object queue.

The required method signature for the name resolution category is defined as:

```
public static Boolean <functionName> (Queue thisQueue,
                                      QueryRequest thisOne);
```

Each name resolution function takes the set of all objects that match any attribute specification. Applied in their preference order, each resolution preference or function is applied to this queue of objects. Each function orders the queue of objects relative to its own internal preference. A `QueryRequest` protocol object that describes the resolution request is provided for verification purposes.

Figure 43 shows the example implementation of the name management function for the single binding attribute. This function implements a verification that the database segment does not contain any objects already bound to the name provided in the `RegisterRequest`. This function takes the queue of objects bound to the specified name as a parameter. If this queue is empty or does not have a value then there are no objects within the database segment that are bound to that name; in this case the binding management may proceed.

The single binding attribute is an example of an attribute that requires the entire database to be examined. To examine the database a request for verification is forwarded upwards through the distributed ORB system. It is possible that during the verification process additional requests for binding or management of the name may be placed at remote nodes. Additional steps must be performed to maintain the coherency of the database segments. When an ORB must examine other nodes it forwards the request to bind to higher level nodes within the hierarchy. A side-effect of forwarding the request is that the name is marked as "bound".

```
public static boolean singleBinding (Queue thisQueue,
                                     RegisterRequest thisOne) {
    if (thisQueue != null) {
       if (thisQueue.isEmpty()) return true;
       return false;
    }
    else return true;
}
```

Figure 43: Implementation of the single binding name management function.

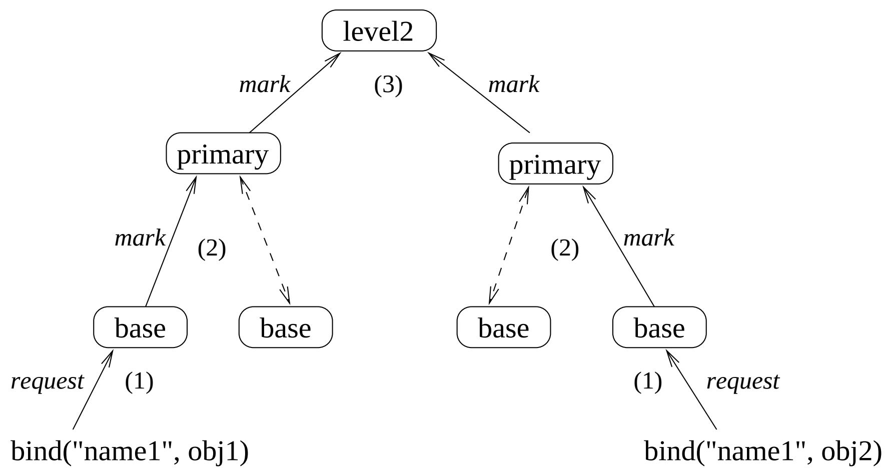


Figure 44: Maintaining database segment coherency.

Figure 44 shows an example of the coherency maintenance process. In this example two requests to bind an object to the name *name*1 are made to different base-level ORBs within different registry domains. A naming model that exhibits the single binding attribute requires that the entire database be examined before the binding management can be declared successful. Stage (1) of this figure shows the concurrent arrival of these two bind requests. As each of the requests is processed by their respective base ORBs, the database segment for that name held at each ORB is marked as “bound”. The process is repeated, (2), as the two requests are forwarded upwards through the hierarchy. At some point within the hierarchy the two requests will be processed by the same ORB node, (3). Operations on the database are locked, hence one request to verify bindings of the specified name will proceed before the other. The first request to be processed will succeed in its binding management.

After a successful binding or name management, a “successful” response is forwarded back down the hierarchy to the initiating ORB. An “unsuccessful” message is forwarded to the rejected second request. At this stage, both subsets of the node hierarchy are accurate as they both record the name as having been “bound”. The “bound” attribute can be used to determine single binding requirements and can also be embedded with a time-to-live value to determine name reuse properties.

Figures 45 and 46 show the implementation of the name management functions for the name reuse and active binding attributes. Together with the single binding implementation function, these functions make up the formal implementation of the mutability attribute. These methods have the same method signature as the single binding implementation method.

The method shown in Figure 45 must verify that any previously bound objects for the requested name are no longer within the object system. This mechanism can be

```
public static boolean nameReuseBinding (Queue thisQueue,
                                        RegisterRequest thisOne) {
    if (thisQueue != null) {
       if (thisQueue.bound(thisOne)) return false;
       return true;
    }
    else return true;
}
```

Figure 45: Implementation of the name reuse management function.

```
public static boolean activeRebinding (Queue thisQueue,
                                       RegisterRequest thisOne) {
    return true;
}
```

Figure 46: Implementation of the active rebinding management function.

implemented through a time-to-live facility where each binding is marked with a global time that marks how long the binding is to remain valid. In a long-lived system such as DISCWorld this is not a suitable solution as there may be different time-to-live requirements for subsystems and service objects. An alternative to the time-to-live facility is a leasing facility where the object is required to explicitly renew the lease. A leasing facility can be used to support different time-to-live requirements but can not be relied upon to maintain complete leasing information. To support long-lived services, and also verify that the bound object is no longer present within the object system, the entire database must be examined and marked throughout the examination process to preserve coherency.

Attributes that require that the entire database be examined upon each operation are expensive to execute. The two management attributes mentioned in this section, along with the global naming attribute, are the only attributes within the current extended naming model that have this requirement. Global naming can alternatively be guaranteed by requiring that the naming system create names itself; these names can be ensured of global uniqueness through their creation process.

Figure 46 places no restrictions on the reuse of names within the system. A name can be rebound regardless of any current or previous bindings and hence the implementation of the active rebinding management function is simple; the method returns true regardless of its parameters or the current state of the database. For this reason and for its service-based semantic properties, the active rebinding attribute is chosen as the default mutability attribute in the DISCWorld ORB default naming model.

```
private boolean completeBindingFunction(Queue element,
                                       RegisterRequest service) {
  Method eachMethod = null;
  Object[] arguments = new Object[]{element,service};
  for (int i=0; i<binding.length; i++) {
      try {
          eachMethod = bindingClass.getMethod(binding[i],bindingOptions);
          Boolean result = (Boolean)eachMethod.invoke(null,arguments);
          if (!result.booleanValue()) return false;
      }
      catch (Exception e) {
            return false;
      }
  }
  return true;
}
```

Figure 47: Implementation of the complete binding function.

7.2.2 Complete Binding, Management and Resolution Functions

The implementation of the generic naming model requires the implementation of the complete binding, complete management and complete resolution functions. These complete functions take a set of binding, management or resolution functions that represent their category of the naming model and apply these functions in their specified order to the requested operation.

The implementation of the complete binding function is shown in Figure 47. This method must take a generic name binding model specification and apply the corresponding functions to verify each name binding. The implementations of the complete management and complete resolution functions are similar.

The contents of the current naming model are by nature unspecified at compilation and can change during the lifetime of an ORB node. Additionally, even the set of functions that may be selected for the naming model is unbounded and hence new attributes and new binding functions can also be added dynamically during the system's lifetime. To support the dynamic integration and invocation of unknown binding functions, the implementation of the complete binding function relies on Java's reflection support [166]. This support allows the selection of functions that match the specified attribute to be invoked without requiring static specification.

Using reflection, each binding function is invoked in the order of specification within the name binding model. If one binding function fails the binding is not valid within the current naming model and is rejected. If all binding functions succeed then the binding proceeds and operations are performed upon the database as required.

7.2.3 Aliasing Support

The DISCWorld ORB system supports the aliasing attribute at two levels: local and global aliasing. A local alias is one that can only be used within the same ORB and can not be used outside of that ORB. This information can not be spread to other domains and can not be used for remote resolution. This restriction allows an object to define a service-based alias that is only valid within its local ORB. This form of aliasing can be used to restrict service support to certain nodes within an ORB system. A global alias can be used anywhere within the registry system. This information can be provided to other ORB nodes and can be used for remote resolution. Essentially a server object has three naming sets: its bound name, local aliases and global aliases. In practice the bound name of an object can be considered part of its global alias set.

When a server object registers with the DISCWorld ORB system it can specify sets of local and global aliases through its attribute and policy specification. If aliasing is supported then these alias sets will be interpreted by the base ORB and, if global, provided to the primary registry for that domain. If a server migrates, its set of local aliases is moved with it to its new location. This model for local aliasing is similar to that used in Sumatra [1,151].

The alias sets provided by a server object at registration must be verified with the current naming model and must also be consolidated with existing alias sets. For example, object *obj*1 is registered with the database with the following global alias set:

```
obj1 :  "webserver", "frontend"
```

Object *obj*2 then requests registration with the following global alias set:

```
obj2 :  "webserver", "frontendwebserver"
```

In a system that supports multiplicity the alias set of *obj*2 must be combined with that of *obj*1 as they share a common alias “webserver”. The alias sets contained within the database then become:

```
obj1, obj2 :  "webserver", "frontend", "frontendwebserver"
```

In a system that does not support multiplicity the attempted alias binding described above is rejected. In this case the alias sets maintained with the database become:

```
obj1 :  "webserver", "frontend"
  obj2 :  "frontendwebserver"
```

Support for the singularity attribute enforces that aliases are verified throughout the entire database as described in Section 7.2.1.

If the set of all alias sets known to the registry is defined as A, and each subset, or alias set, is defined as a_i and the new alias set is defined as a_{new}, the rules governing alias set combination are as follows:

Definition 7.1 (Alias Set Management Function). A function to manage alias sets is a function, ϕ, for a set of alias sets, A, with subsets $a_i \in A$ for $i \in 1..n$, can be defined as a function $\phi : 2^A \times 2^A = 2^A$ where 2^A is the domain of all alias sets. For a given alias set a_i, ϕ is defined by the following operations:

1. if $a_{new} \not\subset A$ then
   (a) $A = A \times a_{new}$
2. if $\forall i, a_i \cap a_{new} \neq \emptyset$ then
   (a) $a_{new} = a_i \cup a_{new}$
   (b) $A = A \setminus a_i$

The effect of this function is that if a new alias set is unknown (*i.e.* it contains no known names) then it is to be added to the database of all alias sets. If there exists alias sets within the database that have common aliases with the new alias set then these alias sets must be combined.

Aliases are distributed throughout the registry system using the distribution mechanisms for object information as described in Section 7.1.3. Each registry contains all sets of aliases as defined by its locally registered servers; each primary registry contains all sets of aliases defined by all services within its registry domain. Alias verification and marking is performed depending on the naming model's support for singularity and multiplicity.

When a client requests a service through a name that is an alias (local or global) from its local ORB node, resolution can be performed immediately. If the alias is unknown remote registries are contacted until the name is resolved. When the client's local ORB receives the resolution result it examines the global alias set for any returned object references and consolidates any new information. Future references through the initially unknown alias are now able to be resolved locally.

The alias support integrated into the DISCWorld ORB system takes advantage of the distributed nature of the namespace database. By caching alias set information, name binding information can be spread throughout the system. In a system that supports multiplicity this caching causes aliases to be irrevocable. If two names are linked by an object to be aliases they can not be unlinked during the lifetime of the object system.

7.3 Mobility

The implementation of mobility within the DISCWorld ORB system consists of the implementation of the mobility service management thread, base support classes and APIs, necessary protocol objects, and the relocation mechanisms to maintain reference validity throughout object migration.

7.3.1 Migration Implementation

The model of object migration used within the DISCWorld ORB system is similar to that found in the Ajents [95] system. Using interpreted Java, access to the execution state of objects is limited. This restriction forces a model of weak mobility; *i.e.* movement of object code only.

Migration within a weakly mobile system is often accomplished by checkpointing objects between invocations to provide intermediate views of the state of the object. An object in such a system is only able to migrate between method invocations. This is the model that is used in the implementation of the DISCWorld system. A consequence of this model is that any invocation requests that are received during the migration process must be queued and forwarded to the migrated object after its migration.

The process of object migration is divided into two stages. The first stage is object suspension. Before an object can be migrated it must be suspended from accepting further invocation requests. The state of suspension continues until all previously executing invocations have completed. All queued invocations are recorded in their order of receipt. After suspension is complete, migration of the server object can occur. The second stage in the migration process is the movement of the server object from its current location to its destination location. After both of these stages have completed successfully the object has completed migration.

To facilitate object migration the standard database management interface provides operations to mark a database record as suspended or moving. When a database record is marked as one of these, a request to initiate the requested operation is sent to the server object's migration fragment adapter. The object's adapter is then responsible for the tasks of suspension and movement.

Any server object within the DISCWorld ORB system can use the mobility support services if the object supports a serialisation mechanism. Serialisation mechanisms are used to translate an object into a byte form that can be translated back to the same object form. The DISCWorld ORB system provides base classes that can be used to develop mobile server objects. These base classes automatically access the appropriate serialisation and migration protocol objects during a migration.

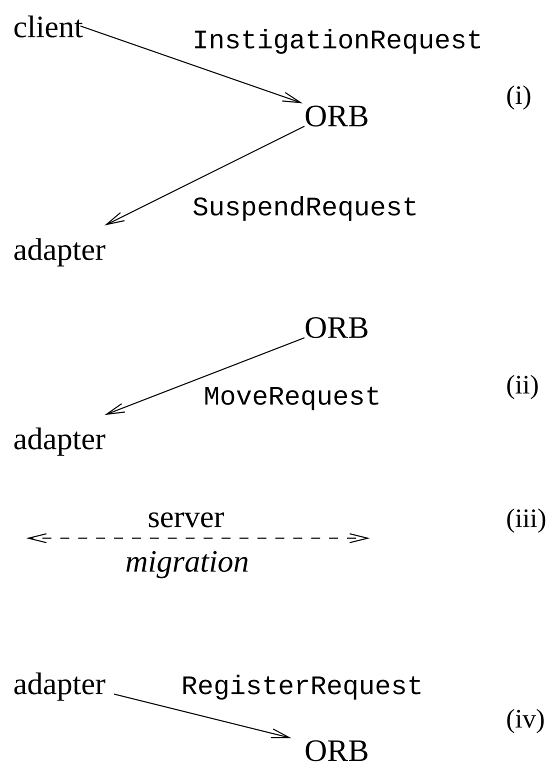


Figure 48: Object migration protocol.

7.3.2 Migration Protocol

Object migration can be the result of either autonomous migration or ORB-induced migration. A mobile object can request migration in response to environmental changes or requirements. Migration can also be requested by the ORB system to perform such tasks as load balancing. Both of these forms of migration are handled using the migration protocol object in the DISCWorld ORB communications protocol.

Figure 48 shows the four stages involved in migration using the migration protocol. Figure 48 (i), shows a client sending a request for migration instigation to a local ORB node. This protocol object invokes the database operations to suspend and move the mobile object. After the ORB processes this request, it then sends a request to suspend the server object to the server object's migration adapter. Following this request, the ORB then sends a request for the server object to move (ii). Figure 48 (iii) shows the third stage in the migration process; the actual object migration. After migration, the server object's migration adapter sends a request to a local ORB to reregister with the ORB system.

Figure 49 shows an example implementation of the migration initiating protocol object. The process method for this protocol object requests the appropriate operations from the database. The invocation of the `suspend` operation causes a `SuspendRequest` protocol object to be sent to the migrating object's migration fragment adapter, followed by a `MoveRequest`.

```
public void process(RegistryData ds, java.io.ObjectOutputStream out)
                    throws ResponseCorruptionException {
    ResponseObject response = new ResponseObject();

    try {
        ds.suspend(this);
        out.writeObject(response);
    }
    catch (IOException e) {
           throw new ResponseCorruptionException(e);
    }
    catch (Exception e) {
           obj.setAchieve(e);
    }
    ds.move(this);
}
```

Figure 49: Implementation of the migration initiation protocol object.

```
public void process(Object service, protocol.AdapterThread adapter) {
    adapter.suspendServer();
    adapter.suspendFragments();
}

public void process(Object server, protocol.AdapterThread adapter) {
    if (adapter.ready())
        adapter.move(destination,port);
}
```

Figure 50: Process methods for the suspend and move protocol objects.

Upon their arrival at the migrating object's adapter these protocol objects perform the tasks of suspension and movement.

Figure 50 shows the process methods for the `SuspendRequest` and `MoveRequest` protocol objects. These protocol objects have a slightly different process method to that used to process database operations. The first process method shown in this figure performs suspension by requesting that the object's adapter suspend both the server object and all migration fragments. The second process method is that used to move the object. When the adapter has successfully suspended all objects it is then able to move.

The `move` method invoked by the second process method in Figure 50 serialises the server object and packages it in a `MobileRequest` protocol object. This protocol object, as shown in Figure 51, creates a new object from the serialised byte stream, or its class representation if serialisation could not be performed, and re-registers the mobile object with its new database segment.

```
public void process(ObjectOutputStream out) {
    if (mobileObject instanceof java.lang.Class) {
        try {
            Class tmp = (Class)mobileObject;
            mobileObject = tmp.newInstance();
        }
        catch (Exception e) {
              ResponseObject robj = new ResponseObject(e);
              out.writeObject(robj);
              return;
        }
    }

    // Register the migrated object with the new database segment.
    int port = Registry.register(mobileObject,policy,true);

    ResponseObject robj = new ResponseObject();
    out.writeObject(robj);
}
```

Figure 51: Implementation of the migration protocol object.

The full migration process supports external requests for migration of a remote object. Subsets of the protocol can be used to support other forms of migration. For example, a migration requested by the ORB system does not need a protocol object to perform operations on the database segment. An autonomous migration has to inform the ORB system of its migration but does not require the transmission of a `SuspendRequest` or a `MoveRequest` from the ORB system to itself. An autonomously mobile object can invoke the operations required within its migration fragment adapter through the `Registry` API.

7.3.3 Relocation Implementation

The relocation mechanism used within DISCWorld is an explicit update mechanism. Update messages are sent between migration fragments and client interface fragments when a server object migrates. It is only after the migration process has succeeded that update messages are sent. This delay is chosen to limit the requirements of migration rollback. The updating of clients is performed in the final stage of the migration protocol described in Section 7.3.2. In this stage the actual migration is performed; once the migration is complete, the migration fragment adapter updates all of the connected clients.

Figure 52 shows the implementation of the `move` method in a migration fragment adapter. This method is responsible for completing the migration protocol by sending a `MobileRequest` object to the migration destination. This method is also used to finalise migration details and to perform the relocation requirements. If migration is successful, the `move` method proceeds to send an `UpdateRequest` object to each of the connected

```
public void move(InetAddress destination, Object server) {
    MobileRequest mobile = new MobileRequest(server);

    Connection dest = new Connection(destination);
    dest.send(mobile);
    ResponseObject response = (ResponseObject)dest.receive();
    // Perform migration and verify success.

    if (response.getAchieve()) {
        for (int i=0; i<children.length; i++)
             children[i].send(new UpdateRequest(response.getResult()));
        // Update protocol objects are forward to each connected client.

        Registry.deregister(service);
        // Deregister service from current database segment.
    }
    else reportFailure(robj);
}
```

Figure 52: Implementation of the update relocation mechanism.

clients. Each connected client has a corresponding migration fragment that can perform this transmission on behalf of the adapter.

Each `UpdateRequest` object contains the new location of the mobile object and the token identifying each client. The `UpdateRequest` object sent to each client is intercepted by the client's client interface fragment for the specified server object. This client interface fragment updates its cached location hint information and is able to reconnect to its newly migrated migration fragment. If reconnection is unsuccessful, the client interface fragment can repeat resolution through the ORB system.

7.4 Communication Models

The DISCWorld ORB system supports two forms of remote client/server communication: RPC-style communication and agent-based mobile communication. The RPC-style communication model fits naturally into the DISCWorld ORB system due to the system's support for the fragmented object model. Support for mobile communication objects also fits naturally in the DISCWorld ORB model due to its support for a global namespace, migration and relocation transparency.

7.4.1 RPC-style Communication

RPC-style communications models involve the communication of requests between two distributed object fragments. A client-side fragment acts as a proxy for remote communications. This proxy has the same method signatures as the remote object and is able to translate local invocation requests into remote invocation requests on the server-side. A server-side fragment receives these requests, invokes them on the server object and returns the result to the client-side fragment. The main benefit of RPC-style communication is that, by using a proxy, the client invokes remote operations as if they were local. Multiple client interface fragments can be associated with each server object to provide support for different usages of the available communication models.

The client interface fragments and migration fragments used in the DISCWorld ORB system to support relocation transparency are also used to support RPC-style communication. The client interface fragment acts as the client-side proxy while the migration fragment acts as the matching server-side fragment. To enable this the client interface fragment must have the same method signatures as the server object.

The DISCWorld ORB system extends the client interface fragment to support multiple communication models; namely synchronous, asynchronous and future-based communication. When creating a client interface fragment to match a server object, the server object is able to specify which methods should controlled by which communications model. This specification is done by annotating an interface that represents the publicly available methods of the server with keywords that describe the required communications model. The keywords used in the DISCWorld ORB system are `asynch` and `future`. By default, methods are synchronous. An example of an interface written in this Java-based pseudo code is shown in Figure 53.

A synchronous method implemented in this manner must translate any invocation parameters into a transmittable form, send the invocation request and wait for a response. This response contains any result returned by the remote invocation. An asynchronous version of this process embeds the response waiting in a separate thread, an `InvokeThread`, using Java's multithreading support [134]. This thread can then be monitored for success or failure of the invocation.

The implementation of `future` objects is performed using the same threading mechanism. In an asynchronous and future-based method the result of the method call will reside in the secondary thread until it is required by the client. Attempts to access the `future` object

```
public interface ServerInterface {
    asynch public future String getHostName();
    public void giveHostName(String hostName);
    asynch public void closeServer();
}
```

Figure 53: Annotated server interface definition.

before the remote method completes results in the client blocking until the result is ready. To prevent unnecessary blocking, asynchronous and future-based remote methods can be polled to obtain their completion status.

7.4.2 Mobile Communication

The dynamic creation of mobile communicator objects is performed through the provided APIs. These APIs provide methods to dynamically create a mobile object and assign it a series of itineraries and tasks. Tasks can be linked, with the result of one task being passed as a parameter to future tasks; parameters can also be provided by the creator of the communicator. Each mobile communicator is named within the object system by a location independent name that can be used as a reference by clients to remotely monitor and invoke methods provided by the communicator. The API that provides these methods is presented in Appendix B.

An itinerary corresponds to the name or attribute-based description of a server object. A task corresponds to a method to invoke on that server object. Each mobile communicator maintains a queue of itineraries and tasks to perform for each itinerary. This queue can be traversed multiple times to allow the communicator's actions to be repeated at the request of clients. A communicator's current position within the queue specifies the next migration or invocation to perform. When this object is an itinerary, the communicator requests a reference to the specified server object through the ORB system. Once a reference has been found the communicator migrates to the location of the server object through colocation. Once resident at the same host as the server, the communicator invokes its tasks locally.

Figure 54 shows the arrival process of a communicator object at a new location. Once arrived, the mobile communicator is registered and, if tasks are still to be invoked, it continues execution. The communicator will continue to invoke tasks on its local server object until either a new itinerary is the next object in the queue, or the queue is completed. If a new itinerary is specified, the communicator commences the process of resolution and colocation.

```
public void process(ObjectOutputStream out) {
    int port = Registry.register(mobileObject,policy,true);
    MobileCommunicator mobj = (MobileCommunicator)mobileObject;
    if (!mobj.Completed().booleanValue()) {
         mobj.run();
         mobj.send();
    }
}
```

Figure 54: Implementation of the mobile communicator migration process.

It is possible that during a colocation, the remote object that the communicator is being colocated with may choose to migrate. To prevent communicators continually chasing frequently migrating objects throughout the object system a limit can be placed on the number of colocations allowed per itinerary. If the limit is exceeded invocations will proceed remotely. This mechanism can also be used to support the dynamic creation of stationary agents by setting a colocation limit of 0.

7.5 Summary

The implementation of the prototype DISCWorld ORB system includes an adaptive ORB. ORBs exist within a hierarchical structure designed to support a distributed global namespace. The positioning of ORBs within the hierarchy is performed dynamically and adaptively. Adaption of the ORB structure within a registry domain is performed in response to partial node failure within the hierarchy.

Each ORB manages a segment of the global namespace using the generic naming model. The support for a generic naming model includes a framework for the development of new attributes and preferences as extensions to the naming model defined in Chapter 5. Implementations of complete binding, complete management and complete resolution functions have been developed. These implementations are designed to support a dynamically specified naming model consistent with the specified framework.

Distribution of information and objects within the DISCWorld ORB system is performed using a combination of consistent registry domain information and cached administrative information. Algorithms to maintain coherency within the database are also defined. A protocol to support communication between the distributed object system components has been defined. These protocol objects are self-evaluating and a framework for the further development of protocol objects is introduced.

The implementation of the mobility system involves the development of specialised protocol objects to support object suspension and migration. The full migration protocol is designed to support migration of server objects as requested by an external object. Subsets of this protocol can be used to support ORB-requested migration and autonomous migration.

The implementation of the communications models used within the DISCWorld ORB system relies on existing models and implementation. The implementation of RPC-style communications uses the fragmented object model designed to support relocation transparency. The mobile communicator system uses the native mobility support and the distributed naming system to perform optimised semi-autonomous communication.

Chapter 8

Evaluation

The evaluation of a system such as the DISCWorld ORB system must consider both qualitative and performance-based evaluation. The system must be evaluated with respect to how well it satisfies the requirements of a transparent mobile object system and how well it supports a flexible naming model and hierarchical structure. The limitations of the system must also be considered. The proposed model of relocation transparency must be compared both qualitatively and quantitively against existing models.

The performance of the system is also important. To manage a distributed namespace effectively, parameters that define the appropriate scale of the distributed system must be developed and analysed. These parameters must define the optimal size of registry domains and namespaces, and the factors affecting these parameters. It is important to analyse the cost of the location and relocation mechanisms and their transparent nature. The scalability of the system in terms of the number of objects within the system, the number of nodes within the distributed namespace and the frequency of communication must also be examined. This examination must take into account the cost of the underlying communications protocol.

Qualitative analysis is the analysis of the quality of an entity with regard to the requirements the entity is attempting to meet. A qualitative analysis of the DISCWorld ORB system must evaluate the naming and transparency models used within the system and also the application of those models in the implementation of the system.

The DISCWorld ORB system is designed to support a widely distributed object system where objects are named within a global namespace and can be mobile. Names are modelled as location independent and transparent references that support relocation transparency using a combination of an update model and a distributed registry system. The naming model used within a distributed system has great impact on the operation of the system, affecting both the cost of managing the system and the semantics supported by the system. The DISCWorld ORB system is designed to support extensible and flexible naming models.

The concepts of extensibility and flexibility within the model have been extended to the design of the underlying communications protocols.

To measure the performance of the DISCWorld ORB system the actual costs of location and relocation within the system must be measured. Additionally, costs involved in the communications protocol must be examined as well as costs involved in the operation of the ORB system. To explore the optimal size of a registry domain, the effect and load of each request on the system must be measured. The optimal size of a registry domain is affected by the database segment size that can be easily supported by a base ORB and also a primary ORB. The scalability costs for the distributed hierarchy must be measured.

The scalability of a distributed object system has been defined as a cost of $O(n)$ for including an object into a system (for a number of objects within the system n); and a cost of $O(log\ n)$ for the maximal performance degradation [10]. The models for location and relocation transparency can be compared in terms of their scalability. Scalability is evaluated with regard to the number of clients, nodes and servers within the system.

The environment for the performance measurements is a farm of Digital AlphaStation 255/300 MHz machines, connected by 155 Mb/s ATM. Additional measurements were taken on a Sun Ultra2 (dual processor 168 MHz UltraSPARC), a Sun E250 (dual processor 300MHz UltraSPARC) and a Sun E450 (four 300MHz UltraSPARC processors) connected by 10Mb Ethernet. The versions of the Java Development Environment (JDK) used were JDK 1.1, JDK 1.2 and JDK 1.3. These versions provide full support for multithreading, serialisation and reflection. The tests were performed using the adaptive distributed web server (described in Section 6.1.2) as an example application.

Multiple environments are used for testing to ensure that the behaviour of the system components is consistent regardless of the implementation platform. The different implementations of a single JDK version provide different error conditions for common socket events and failures [52] and hence these different conditions must be trapped and handled in a consistent fashion.

8.1 ORB System Design

ORBs within DISCWorld have been designed to support multiple ORB-based services and to cooperate with other ORB nodes as part of a distributed hierarchy. This design has support for fault resilience in the form of ORB adaptiveness in response to node failure. ORBs are able to reposition themselves to higher positions within the hierarchy and take over the position of a primary registry if necessary within an existing registry domain.

8.1.1 Distribution

The distribution of the ORB system is performed using a hierarchical structure. Nodes exist within either the active or administrative node layers. The global namespace is distributed in hierarchically structured segments over all nodes within the ORB system; both active and administrative nodes are used to manage the namespace database. These layers are used to separate the management of the distributed namespace into the forms of a "push" model and a "pull" model. Nodes within the active layer are organised into adaptable registry domains; the information contained within a registry domain is complete for that database segment. Hence, primary registries can always be contacted for up-to-date information on the structure of database segments managed within the domain. The "pull" model applied in the administrative layer allows information to be dispersed throughout the ORB system in response to client and server requests. Information is cached at administrative nodes; this caching can lead to inconsistencies between administrative database segment information and those maintained by primary registries.

By having server information distributed in this way, access to an object that is local can be accomplished efficiently while access to an object at a remote node pays an increased price. This model relies heavily on the principle of locality both in terms of its client access and the mobility of server objects. The combination of "push" and "pull" models is used to support this localisation focus.

Two factors affect the success of such a distribution model: the size of each registry domain and the practical support for the principle of locality. As the size of registry domains decreases, the number of remote accesses through the administrative layer increases. The percentage of client accesses with remote delays also increases, as does the load on the administrative nodes.

As the size of registry domains increases, the number of remote accesses through the administrative layer decreases and, hence, most client accesses can be satisfied within the consistent database segment. As registry domain size increases the management load for the primary registry also increases. This load is dependent on the frequency of access by clients and servers. A system that is infrequently accessed is best suited to this model.

The optimal size of registry domain can be realised given the requirements and expected loadings for a system. The parameters to determine this optimal size are the latency bounds which can be tolerated within one domain, the overall size of the object system, and the frequency of client and server access. For example, a large object system can be easily supported in a system where clients use the ORB system to obtain an initial reference, and then use that reference to obtain further references. A system that supports many clients requiring access to many, semantically unrelated objects may require a smaller domain system. A quantitative analysis of these requirements is performed in Section 8.4.

8.1.2 Adaptiveness

The adaptiveness required within the ORB system also has an effect on the performance and efficiency of the namespace management. Each ORB within the system is capable of repositioning itself to a higher position within the node hierarchy. This model allows base registries to take over the role of a primary registry in the case of primary registry failure, and to rebuild the database segment for that registry domain. A consequence of this mechanism, discussed in Section 6.2, is that registry domains may split in response to this adaptive behaviour. This can lead to a continual decrease in the size of registry domains and an increase in communication through the administrative layer.

Manual intervention can be used to partially monitor this situation. If it is detected that the number of registry domains for the same physical node collection is increasing, manual shutdown of the additional primary registries can be used to force base registries to rejoin the initial domain. This is not an optimal solution as it requires external monitoring and intervention. A similar, automated, system can also be used to perform this operation given parameters that define how the system should be structured. These parameters, as mentioned in Section 8.1.1, can be obtained through system measurement and analysis.

8.1.3 Protocol Evaluation

The DISCWorld ORB communications protocol is an extensible protocol based on the use of defined protocol objects. These protocol objects are developed using a framework. Protocol objects are self-evaluating, meaning that they contain the necessary instructions, in the form of operations on a database segment, to perform the action they are requesting.

The alternative to self-evaluating objects is that each ORB or database manager be able to interpret each protocol request. Due to the adaptive nature of the DISCWorld ORB system and the multiple database managers that are used within the system, this model greatly restricts the extensibility of the protocol. The object-oriented approach used in the design of the DISCWorld ORB system, where each class of objects presents a well-defined interface has proved to be successful in the implementation of a prototype distributed object system.

The DISCWorld ORB communications protocol is designed for flexibility and extensibility. These attributes are deemed essential in the development of a prototype object system. The development of a protocol object development framework and the self-evaluation model allows the separation of protocol information from the development of the ORB and database management systems. A consequence of this separation is that protocol objects contain a large amount of information; this information defines how the request will be performed and parameters that the request requires. Hence, the flexibility and extensibility of the communications protocol are implemented at the expense of efficiency.

A major component in each communication within the object system is the transport cost of the appropriate protocol object; the object must be translated into an appropriate byte form, sent to the destination object and then translated back to an executable object. Java provides two mechanisms for translating objects into byte form; the serialisable (using the `java.io.Serializable` interface) and the externalisable (using the `java.io.Externalizable` interface) methods. The serialisable method does not require any input or instructions from the object to be serialised, instead, a generic reflection mechanism is used to translate the contents of each object. The externalisable interface requires explicit instruction from the object to be translated as to which components to translate, and how those components should be translated. Objects translated into byte form using serialisation are larger and the translation process takes longer than those translated using the externalisable method. Object transfer within the DISCWorld ORB communications protocol is performed using the externalisable method. The consequence of this choice is that a more efficient translation method is used but the development of further protocol objects is complicated by the process of externalisation implementation.

Tables 11 and 12 show a comparison of the costs of each of the methods, in terms of the size of the byte stream produced and the time taken to serialise. These costs were measured using JDK 1.2 on an AlphaStation under low system load; the serialisation time was measured by taking the average of 100 serialisation tests. Although there is a dramatic difference in the byte numbers produced, there is only a relatively small difference in the actual cost of the transfer for each class of object. The cost of transferring over a high latency network is related more to the size of the byte array than the initial production cost and, hence, it is more desirable to reduce the number of bytes produced and gain a small production benefit than simplify code production.

Tables 11 and 12 show that there are significant differences in the costs to transmit different protocol objects used within the communications protocol. For example, a `RegisterRequest` protocol object takes almost twice as long to transmit as it does for a `InvokeRequest`. Different protocol objects require different information to be stored; this information can vary the cost and hence the performance of a request greatly.

The DISCWorld ORB communications protocol is designed to support flexible and extensible protocol design. Protocol objects can be added to the system to extend the

Protocol Object	Bytes (S)	Bytes (E)
Register	412 bytes	177 bytes
Invoke	193 bytes	73 bytes
Query	263 bytes	130 bytes

Table 11: Size of protocol objects produced by the serialisation mechanisms.

Protocol Object	Write (S)	Write (E)	Read (S)	Read (E)
Register	505.8 ± 32.11	432.6 ± 16.34	154.4 ± 7.92	110 ± 7.1
Invoke	332.2 ± 8.52	271.4 ± 15.38	150.6 ± 2.88	102 ± 9.05
Query	357.8 ± 9.75	303.4 ± 11.30	133.4 ± 3.91	103.2 ± 1.09

Table 12: Serialisation costs for protocol objects.

interface provided to client and server objects without having to alter the multiple database managers present within the system. The cost of this flexibility is quite high: some protocol objects can take up to 500 milliseconds to transmit between local nodes. During prototype system development and the process of further integration into DISCWorld and DISCWorld's other services (scheduling, bulk data transfer) this protocol design is of benefit as it allows the protocol used to change dynamically and without major system changes. In a full development system, this issue would have to be revisited.

8.1.4 The Hidden Costs of Multithreading

Each ORB consists of a database manager and several service management threads. These threads exist as part of one multithreaded process that utilises Java's support for multithreading and priority-based scheduling [134]. The performance of thread creation and swapping within each ORB is dependent on the support and efficiency provided by each particular platform's implementation of the JDK.

An ORB will usually have four or five service management threads, which may themselves create threads to handle communications. The implementation of the ORB system uses a pool of threads to which communications can be allocated. This mechanism decreases the need for additional thread creation. Each server object's migration fragment adapter is capable of creating an unlimited number of threads. A particular server object can choose to limit this by limiting the number of clients that it will concurrently accept connections from.

Figures 55 and 56 show the costs of thread creation and thread switching within commonly used JDKs on the platforms mentioned at the start of this chapter. These tests were performed 100 times, with the costs averaged to produce those shown in these figures. Figure 55 shows the cost of creating multiple threads with overhead increasing as more threads are required. The creation costs on the Alphas are much higher than the implementation on the SPARCs. Later versions of the JDK perform better than earlier versions in both the SPARC and Alpha cases. For small scalability requirements the later JDK version on the SPARC platform performs particularly well.

Figure 56 shows the cost of switching multiple threads versus the number of threads requiring to be switched, exploring the scalability of thread switching under Java. This figure shows the time taken to perform a given number of thread switches. Each individual thread relinquished control immediately after access, so that the executing thread was switched immediately back to the main thread. In Figure 56, the scalability for thread switching on the Alpha JDKs is very similar, showing no performance improvement with the later release. In comparison there is a marked performance increase between the two SPARC JDKs. The cost of thread management within Java JDKs is discussed in more detail in [96].

Thread usage is kept at a minimum within client interface fragment code as the costs of thread creation and thread switching are highly dependent on the implementation platform and the version of the JDK used. It can not be guaranteed that a client has a JDK version suitable for extensive multithreading. Multithreading within client interface fragments is used to support asynchronous and future-based communication only (as described in Sections 6.6 and 7.4). Multithreading is used in these cases in order to enable clients to concurrently perform remote communication and local computation. Each server object specifies which of its methods will support which communication model (*i.e.* synchronous, asynchronous or future-based). A server object is able to provide multiple interfaces, of which some can be linked to an attribute that defines a suitable JDK version.

8.1.5 Summary

The DISCWorld ORB system uses a combination of "push" and "pull" information flow. This combination enables the system to share information, specifically the global namespace, throughout the ORB system without requiring a centralised or fully replicated database. Registry domains are used to separate areas of replicated information from areas of cached partial information. The optimum size of a registry domain can be decided upon based on the frequency of communication and the size of the namespace.

The design and implementation of the ORB system is focused on adaptiveness and flexibility. These issues enable the development of an extensible object system where protocols and services can be changed and added dynamically. A consequence of this extensibility is the additional overhead of protocol evaluation and service access. Part of this cost is involved in translation mechanisms and multithreading overhead.

The implementation of the ORB system in Java has also lead to performance problems. The implementation of the communications protocol using Java enables platform independent development and access to provided libraries for serialisation, however, it has been shown that the performance of protocols implemented in interpreted Java is equivalent to the cost of compiled C code on four-year-old hardware [111].

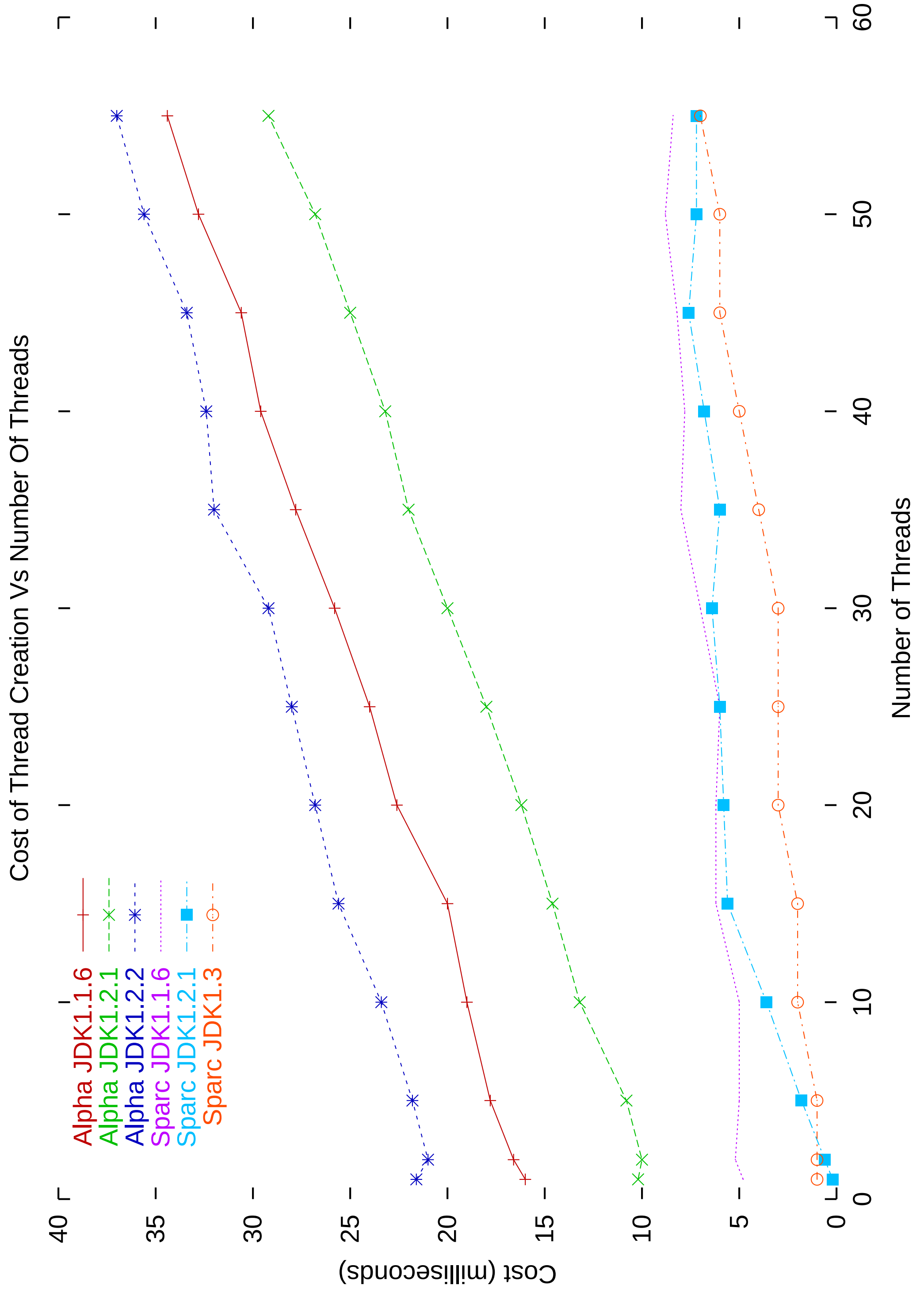


Figure 55: Thread creation costs.

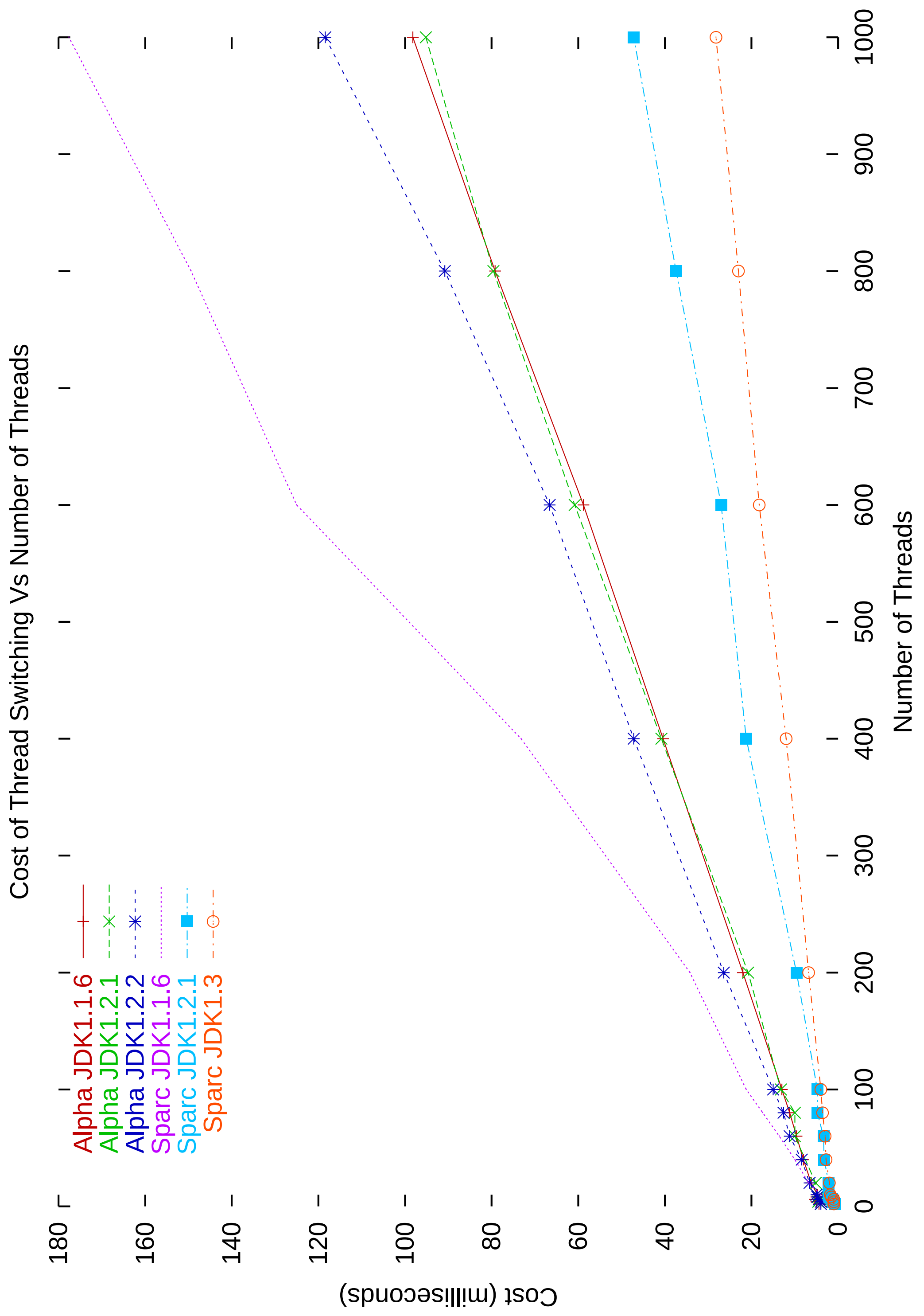


Figure 56: Thread switching costs.

8.2 Communication Models

The DISCWorld ORB system provides two communication models: an RPC-style model and a mobile communicator model. The RPC-style communications model provides synchronous, asynchronous and future-based communications using a fragmented object model. The mobile communicator model provides an agent-based communications model designed to minimise latency between the client and server objects.

The mobile communicator model is designed to support clients without local access to the ORB system and also to support efficient communication between access points that are distant from the required services. Mobile communicators provide an itinerary-based interface that supports semi-autonomous migration; an itinerary-based interface is commonly used in dynamic agent systems [115]. This interface differs from other itinerary-based systems in that it allows results from method invocations to be used as parameters to other, later, method invocations. Mobile communicators also make use of the ORB interfaces and transparency to provide a remote service that can be invoked and repeated at regular intervals. This form of communications is particularly suitable for a system such as DISCWorld, where the collection of new satellite and imagery data is a regular event.

Mobile communicators are able to cope with partial failure. If a communicator is unable to locate a service or complete an invocation it attempts to proceed with its remaining tasks. If these tasks depend on the results of the failed invocation the communicator halts. These failures can only be detected by the client if it is monitoring its communicator, or if it has programmed its communicator to report failures.

8.3 Naming Models

The naming models developed in this thesis rely heavily on the name binding and name resolution models developed by Bayerdorffer [15, 16] and Bowman *et al* [26]. The initial model defined by Bayerdorffer presents attributes used to classify concurrent systems. Bowman *et al* define a model for name resolution based on defined preference orderings. These preferences describe both database and client ordered preferences. The model defined by Bowman *et al* is not a complete model but is a framework for the development of preference approximation functions to appropriately describe system semantics. Extensions appropriate to the classification of distributed mobile object systems are defined in Chapter 5.

The extended naming model is used to classify existing mobile and distributed object systems. These classifications take into account the binding, management and resolution models for each system and are designed to classify mobile object systems with defined naming systems. Although the extended models have been defined for mobile object systems with transparency requirements, subsets of the naming system may be used to classify other forms of object systems.

Distributed and mobile object systems that provide location independence and transparency require that attributes defining this independence and transparency be included in the naming model. To classify object systems without these requirements these attributes can be left out. The locality-based preferences are defined to classify the distributed nature of resolution within an object system. For systems that do not have support for distributed references or distributed resolution these preferences need also not be considered.

The DISCWorld ORB system's support for a generic naming model allows the specification of the required naming model to be performed at system creation. The generic naming system allows the selection of management and binding attributes that are difficult and expensive to execute in a distributed global namespace. These attributes, as described in Section 7.2.1, require that the entire database be examined for each management or binding request. As described in Section 8.1.1, the cost of remote access is greatly influenced by the scale of registry domains used within the system and the frequency of client and server access. The effect of attributes that require full database examination is influenced more by the cost of server access than client access, as the management and binding attributes in question are invoked by server objects.

It is possible to define a naming model that performs inefficiently. These models benefit from a centralised database but are restricted by scale and bottleneck issues that accompany a centralised database solution. Hence these naming models are more suited to small scale systems that can be managed by a logically centralised database in the form of a single registry domain, or by the combination of only a few registry domains to restrict remote access requirements.

The DISCWorld ORB system defines a default naming model that can be used to support transparent and independent naming. The default naming model is described in detail in Section 5.4.2. The default naming model is designed as a naming model well suited to the distributed database model with the restrictions highlighted in Section 8.1.1.

8.3.1 Cost of Coherency

All bindings for a name need not necessarily be managed within the same database segment. For some naming models individual segment management is a sensible solution as additions and removals of database information are not dependent on information held at other nodes. For other naming systems the entire database needs to be examined for each database change. Although the distributed ORB system supports both local and global access to the database, localised access is less network costly.

Within a registry domain the database information is consistent. Within the administrative layer, information is cached and can not be relied upon. To obtain correct information, an administrative node is able to request correct database information from each of the database segments that it is managing. Multiple requests relating to the same name binding may be received at different nodes within the object system. Within particular naming models these requests will interact and possibly conflict. A mechanism for maintaining database coherency is presented in Section 7.2.1. This model requires that each naming model be marked with attributes such as "bound". These markings can be piggy-backed on the requests for validation that must be transmitted to support the naming models.

The coherency maintenance model is quite simplistic and is designed within the limits of the extended naming model. For example, to specifically support naming models based on a single binding attribute a name is marked as "bound" the first time it is bound to an object. The name remains in this state for the system's lifetime. Information about the objects bound to the name is passed throughout the system in accordance with the remaining attributes within the naming model. In a name reuse model, information must be maintained about the "bound" nature of each name and also the existence of objects bound to each name. This information may be dynamically obtained at the time of reuse request.

The coherency maintenance model is not designed as a general model. Instead, the model is designed to operate within the limitations of the extended naming model and the structure of the DISCWorld ORB distribution hierarchy model.

8.3.2 Summary

The naming model developed in this thesis enables a naming system to be chosen based on defined characteristics. The formal definition of this model enables an instance of the naming model to be designed separately from the remainder of the system. This mechanism enables an object system to define and change its naming model without having to alter other components of the ORB system.

Specialised algorithms are designed to operate within the bounds of the distributed ORB hierarchy in order to maintain coherency within the naming model chosen by the system. The naming model chosen greatly influences the efficiency of coherency management and the cost of object binding and location, with some models more suited to a distributed namespace than others.

8.4 Location Costs

The cost of locating an object, as defined in Section 6.4, consists of the cost of resolving the name or attribute specification and the cost of connecting the resultant reference to the server object to enable invocations.

The home location and forwarding location models do not provide any resource discovery mechanisms, hence the location cost for these models is the cost of connecting to a known home or forwarding object location. The update/ORB model has a combined cost of ORB access to perform resolution and connection costs involving the client interface and migration fragments.

Figure 57 shows the average object location costs found within the DISCWorld ORB system with an increasing namespace size. These costs were obtained through tests on the farm of AlphaStations running JDK 1.2.2 under low system load. The results shown are the averages of measurements performed 100 times each. Similar tests were performed on the SPARC platform, obtaining similar measurement differences. In these tests, client and server objects were positioned within the system to enable measurement of object location for local resolution, domain resolution and global resolution for a level 2 administrative node. The costs of local resolution were measured for a local client and a remote client, but showed no significant difference.

Each base registry within the example was provided with a database segment of the same size. Each primary registry was responsible for managing two base registries and hence had a database segment double the size of that managed by each base registry. Throughout the measurement of the object location costs, each base registry's database segment was increased to evaluate the scalability of the ORB-based location mechanism. Initially each base registry managed a segment of 10 names, with each primary registry managing 20 names. This size was increased incrementally until each base registry managed 250 names and each primary registry managed 500 names. To provide scalability, as defined in [10], access to the namespace must be of $O(n)$, where n represents the number of objects (names) within the system. At each level of access within the DISCWorld ORB hierarchy (local, domain and global) access costs on average are constant up to a namespace segment size of 1000 names. Above this size, the location cost is well within the bounds of $O(n)$ for the sizes of namespace considered.

8.4.1 Effect of Namespace Position

The average cost of locating an object is independent of the size of namespace up to large sizes. It is important to explore whether different regions within the namespace, and hence the hashtable managing the database segment, result in different object location times. Measurements of object location costs were taken at different locations within each database

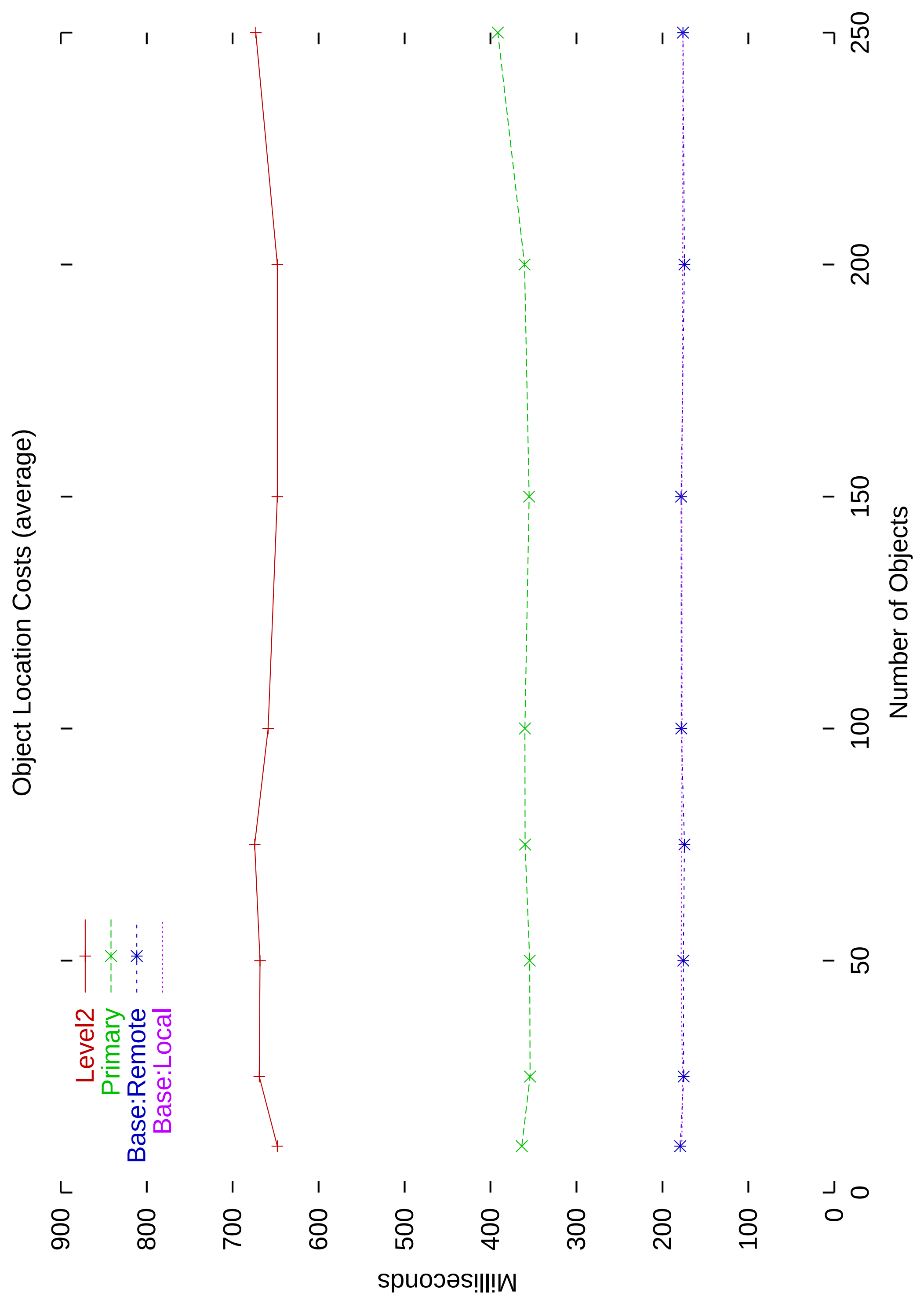


Figure 57: Average object location costs within the DISCWorld ORB system.

segment; for a segment size of S, measurements were taken for locating the objects at the positions corresponding to: 0, $S/4$, $S/2$, $3S/4$ and S.

Figures 58, 59, 60 and 61 show the average location costs for each of these measurements for an increasing namespace size. Tests were performed on a farm of AlphaStations running JDK 1.2.2 and were repeated 100 times and averaged to produce the measurements shown in the figures. Figure 58 shows these costs for a local client locating a server registered on the client's local ORB. Figure 59 shows these costs for a remote client locating a server registered on the client's local ORB. Figure 60 shows these costs for a remote client locating an object contained within the same registry domain. These figures show that for small segment sizes, up to 800 names, the object location costs are consistent regardless of the object's position within each database segment.

The measurements shown in these figures appear to be widely separated, however the range of separation is less than 20 milliseconds for the first two figures, and less than 70 milliseconds in the third figure. The measurements taken in these examinations are roughly consistent and average to almost identical measurements (as shown in Figure 57).

Figure 61 shows the location costs for a remote client locating a server registered on the client's local ORB within a large namespace. For these large segment sizes (between 1000 and 1500 names per segment) the position of the object within the segment becomes an important factor in its location cost. From these results, it is observed that the optimum namespace size for a primary registry's namespace is less than 1000 names. The optimal size for a base registry's namespace is the size of the domain's namespace equally divided amongst the base registries within the system.

The optimal size of a registry domain is affected by the the size of each database segment, the number of base registries, and the frequency of client and server interaction through the ORB system. Figures 57 and 61 show that the size of the database segment managed by a primary registry does not affect the cost of location. An example registry domain system with 10 base registries, each responsible for 80-100 server objects is easily achievable within the performance limits of a single primary registry. This is not an upper limit on the namespace size for each registry but instead represents the point at which resolution increases in cost. In this example, each base registry could be responsible for several physical nodes within the system.

The number of base registries within a domain is dynamically determined. Given an expected number of objects or nodes within the system the latency bound, n, of a domain system can be determined through experimentation.

The frequency of client and server object interaction with the ORB system changes the load, and hence response time, for ORB requests. The ability to service multiple concurrent client or server object requests is controlled by each ORB's ability to create and control its execution threads. These costs, as examined in Section 8.1.4, vary greatly with the platform's support for Java multithreading.

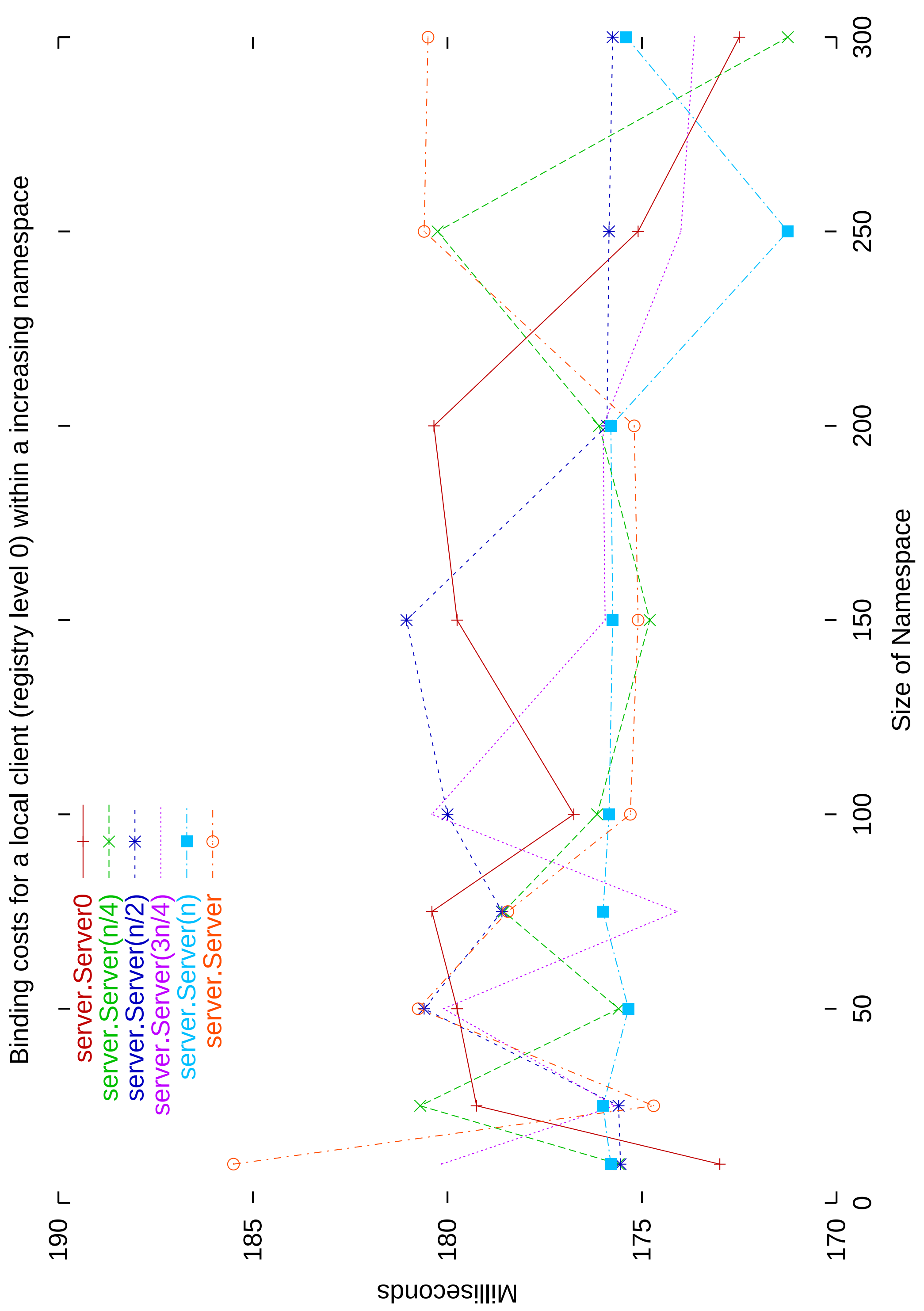


Figure 58: Cost of a local client locating a local server.

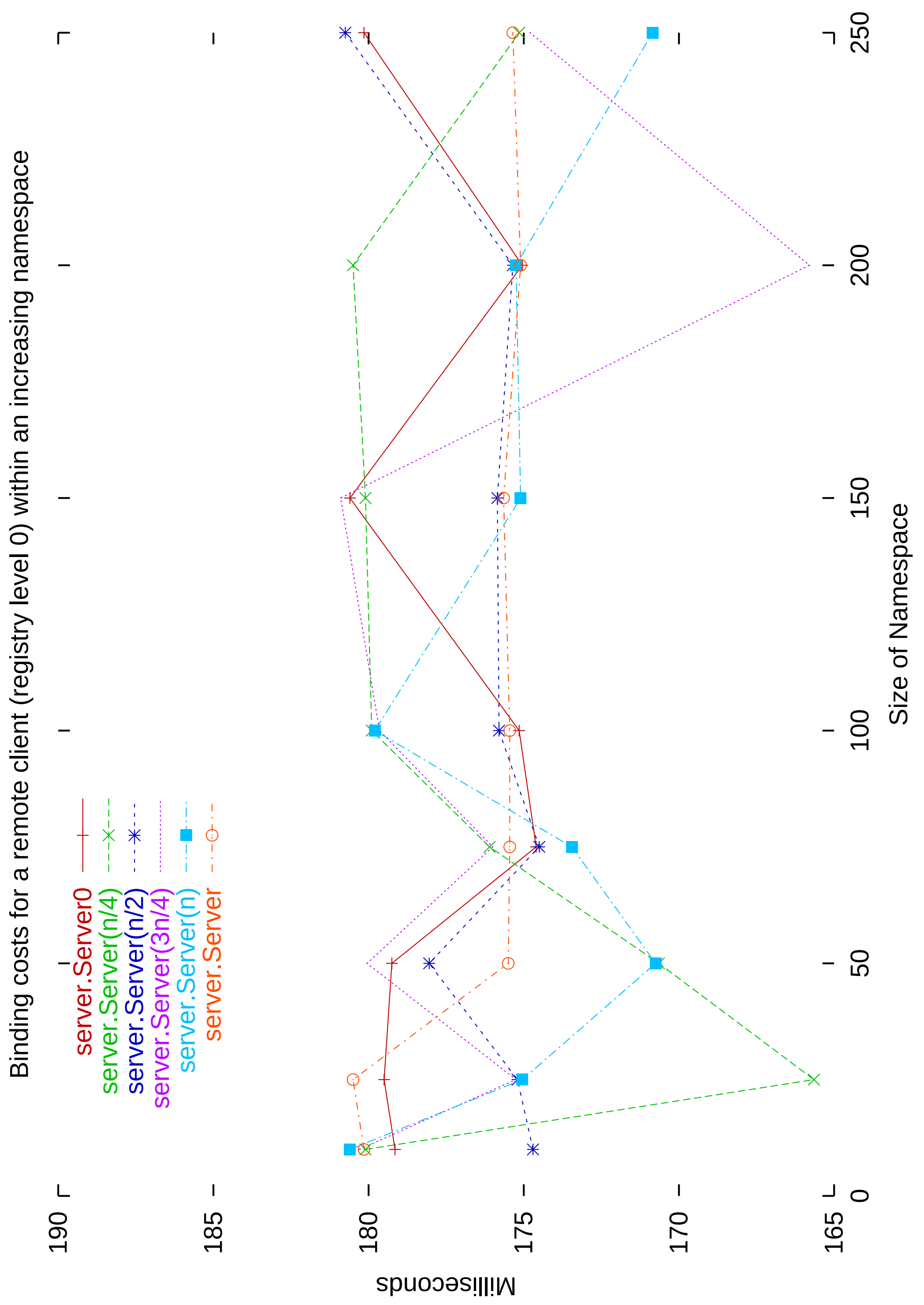


Figure 59: Cost of a remote client locating a local server.

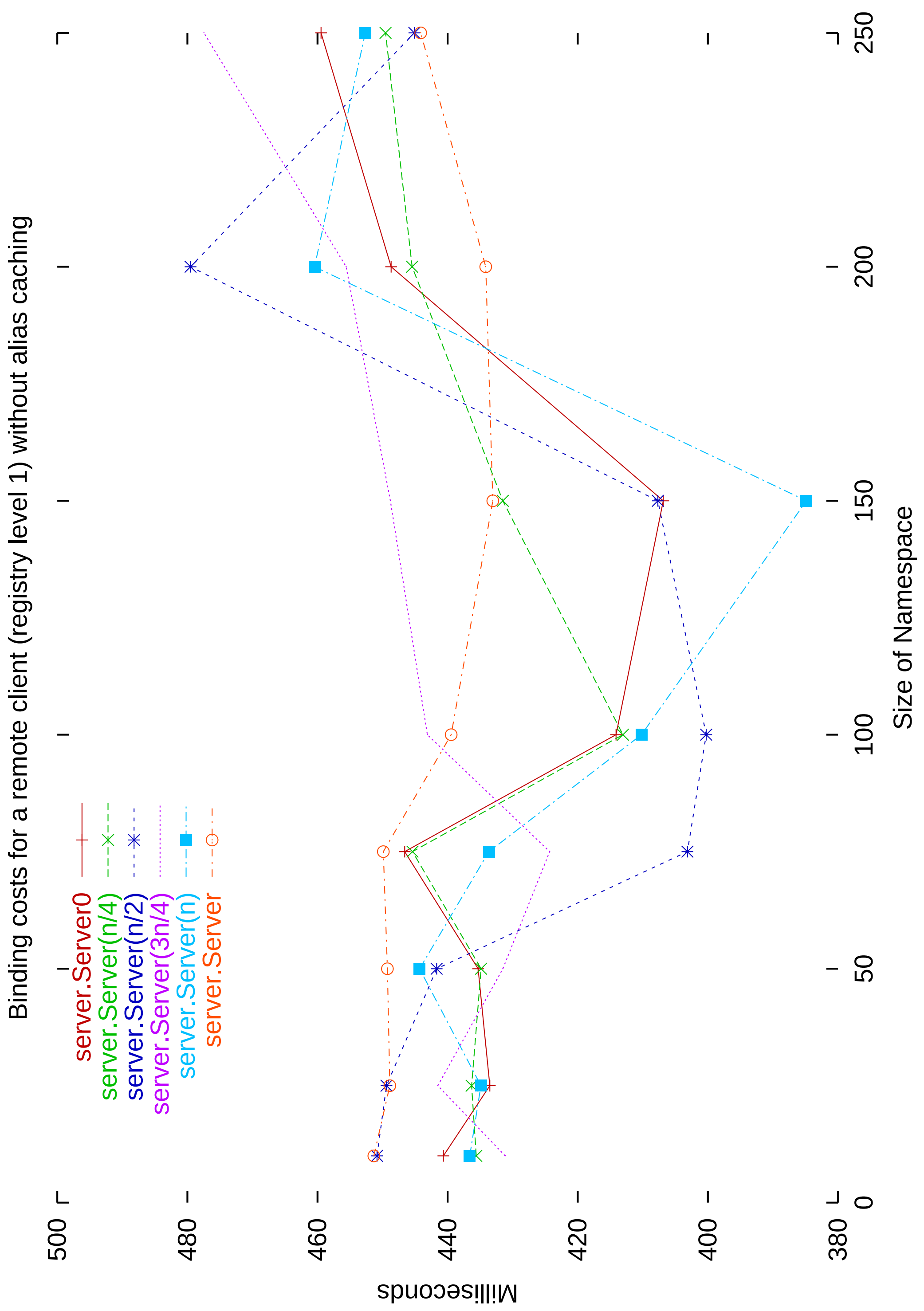


Figure 60: Cost of a remote client locating a domain-based server.

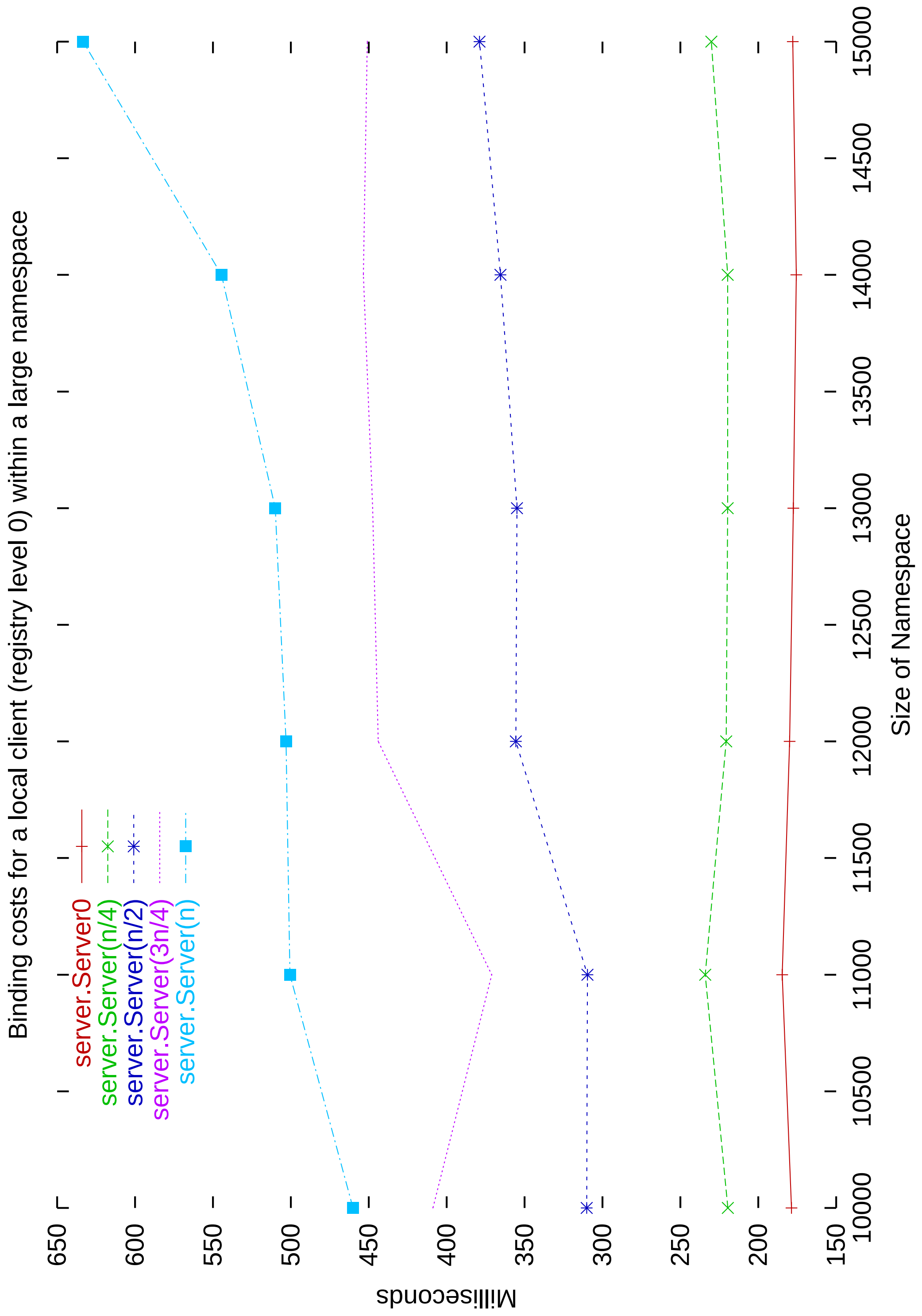


Figure 61: Cost of a remote client locating a local server within a large namespace.

8.4.2 Namespace Dispersion Costs

The cost of data transfer throughout the ORB system is quite substantial. Each request requires the transfer of an appropriate protocol object. These protocol objects may also contain piggy-backed information, for example an alias set, name binding or naming model information. Within the DISCWorld ORB system it is possible to disperse the information contained within each database segment to remote nodes by piggy-backing this information in response to resolution and binding requests. A common form of this information dispersal is to include the alias sets pertinent to the resolution result or binding query. These alias sets can then be cached and used at later points during system operation thereby removing the need for additional remote communication.

Transmitting alias set information places an additional cost on the DISCWorld ORB system. As alias sets expand, through set consolidation, these costs can inhibit the efficiency of name resolution and binding within the system. The effect of alias set dispersal is that, with each remote object location, the located object's alias set is returned to the local ORB. This alias set must then be consolidated with the current alias sets known to the local ORB. When all objects within the system are within the same alias set, all naming information must be returned for each resolution and then consolidated with the known alias information. This is the worst case to be considered.

Figure 62 shows the object location costs for the worst case scenario as well as the object location costs without alias caching. These tests were performed on a farm of AlphaStations running JDK 1.2.2 under low system load; the results shown are the average of 100 measurements of location cost. Location costs are shown for an object within the alias set cluster and for a single object outside of this cluster. The location of objects within the alias set cluster requires transmission of the full alias set from the resolving node to the requesting node. The cost of this location consists of the time taken to transmit the alias set and also the time taken to consolidate the alias sets. The major component in this cost is the transmission cost.

Figure 62 highlights the major performance decrease found for the worst case scenario. Without alias set caching or with small alias set caching (where a small alias set is assumed to consist of 5 aliases) there is no performance decrease as the size of the namespace decreases. Alias set caching and transmission allows alias information to be dispersed throughout the object system and enables the growth of alias sets through consolidation. This information can reduce the number of required remote resolutions by allowing local objects to be recognised as aliases for the requested name.

In Figure 62 the most efficient object location result from a system using alias set caching for small alias sets. This system is more efficient than the systems that use no alias set caching (both within and without the worst case cluster) because the local object is able

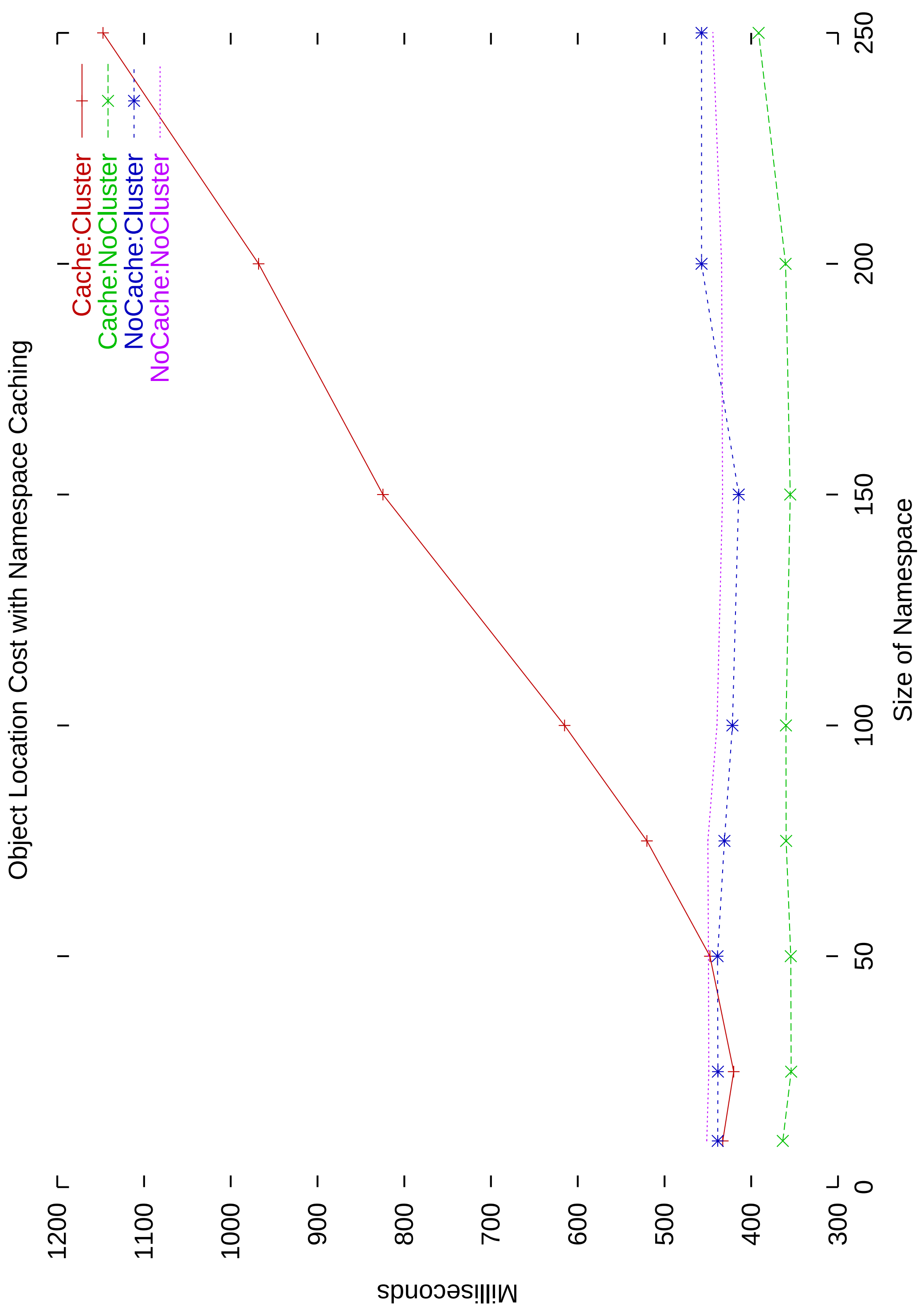


Figure 62: The effect of namespace caching within the DISCWorld ORB system.

to be used in all cases. In the systems without caching, references through aliases always result in a remote resolution.

Transmission of alias sets requires the alias set to be translated into an appropriate byte form, transmitted and then translated back into the correct object or set form. This translation is performed using either serialisation or externalisation. Serialisation allows the alias set objects to be dynamically extended to transport additional information, however serialisation is the least cost efficient of the two translation methods, as seen in Section 8.1.3.

Figure 63 shows the serialisation costs in terms of the time taken to read and write the alias sets in the data transfer. These measurements were taken on an AlphaStation with JDK 1.2.2 under low system load; the results shown are the average of 100 measurements for each data point. This figure shows that the serialisation mechanism scales poorly and is more expensive for both reading and writing alias sets.

Using externalisable objects rather than serialisable objects limits the extensibility and flexibility of the protocol and alias objects used within the DISCWorld ORB system (see Section 8.1.3). When designing extensions to these objects, methods to translate the contents of the extensions must also be implemented. This requirement does not change the effectiveness of separation of protocol and alias implementation from the remainder of the system but does require more work on behalf of the designer of system extensions.

8.4.3 Summary

The location of objects within the DISCWorld ORB system consists of the costs of name resolution within the distributed namespace and the costs of using the fragmented object model for connection. The cost of name resolution within the ORB system is dependent on the client and server objects' positions within the ORB hierarchy. As expected, different, and increasing, costs are incurred for local, domain and global resolution.

This section shows that the cost of location is dependent on the size of the namespace, however, this dependency does not have a marked effect for domain namespaces of less than 1000 names. The measurements described in the section also show that the costs of location are dependent on the name's position within the global namespace. Again, within a domain namespace of less than 1000 names this difference is negligible.

8.5 Transparency

The update model of relocation transparency proposed in this thesis can be evaluated with respect to its support for transparency and location independence and also by its scalability in comparison to other commonly used relocation transparency models. As described in Chapters 1 and 2, the commonly used models are the home location model and the forwarding location model.

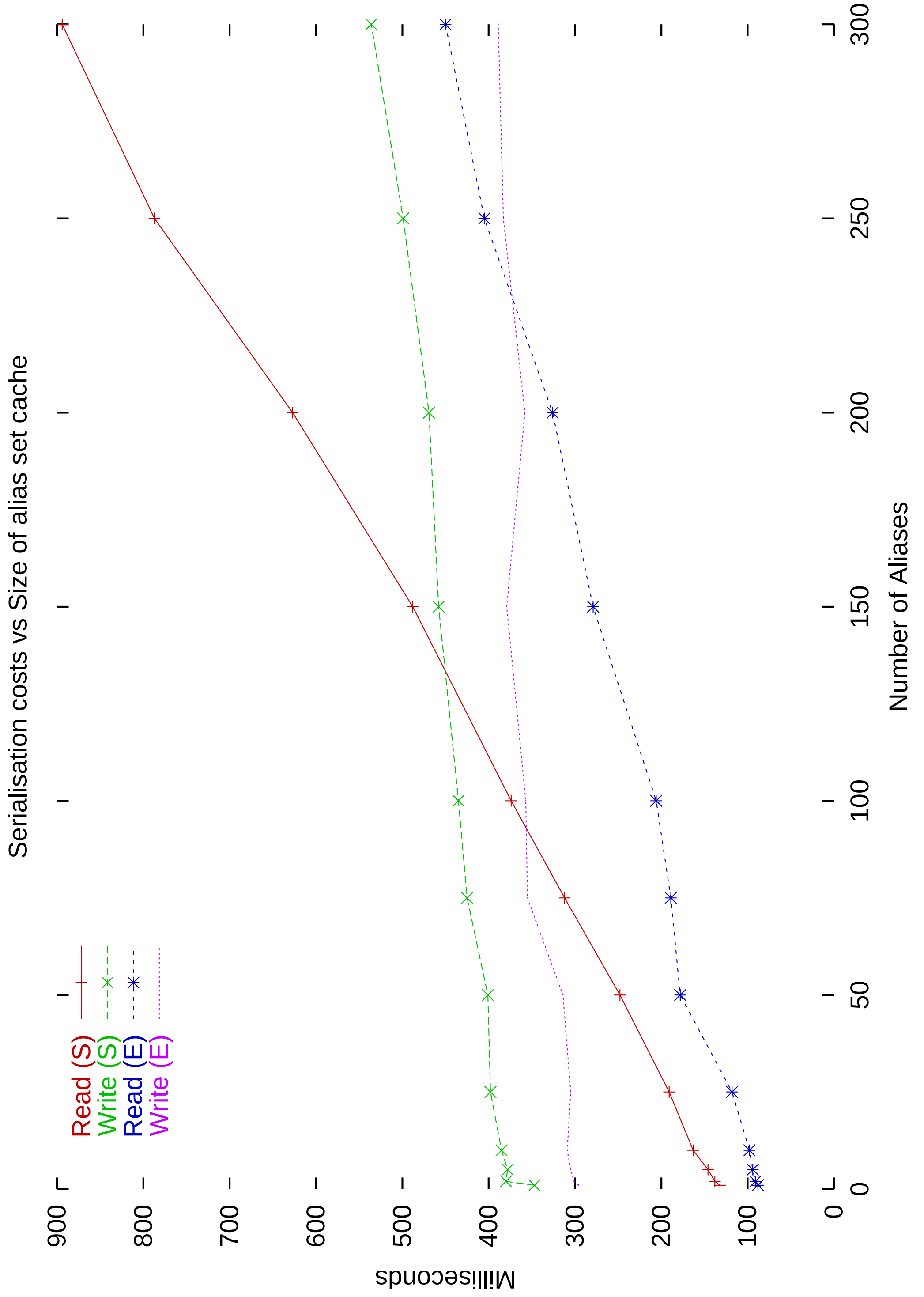


Figure 63: Serialisation costs for alias sets.

The update model requires a duplex channel between itself and each client. These channels connect each client interface fragment with a migration fragment. A migration requires an update message to be sent to each client that is connected; the reliability of message receipt is dependent on the underlying protocol. When messages are lost or become garbled, the update model uses the location transparency mechanisms to relocate the server object. Location transparency is provided by location transparent and location independent references that can be resolved using the distributed ORB system as an access point to the global namespace.

The concept of using multiple mechanisms to provide relocation transparency has been used in several mobile object systems (MOA [131], Charlotte [3, 4] and V-System [124]). There is no system known at the time of writing this thesis that uses a combination of an update model for connected clients and a location transparent distributed naming system. Using multiple mechanisms provides multiple levels of transparency, however, it can also introduce additional communication and work into the system. By integrating the backup relocation mechanism into the object location mechanism no additional work is required. By using an ORB-based abstraction over the distributed namespace, the location mechanism is able to provide resource location and attribute-based searching facilities.

8.5.1 Transparency Models

A comparison of the proposed update model with the home location and forwarding location models must be performed with regard to the cost of location and relocation and also each model's support for scalability. Scalability must be considered in terms of object system, access, server and client scalability. The investigation of the predicted cost models is divided into the categories of local reference, domain reference and global reference. Costs can be categorised into location costs, invocation costs and relocation costs. Only the update model has an explicit relocation cost which is defined as the cost of reconnection and update message transmission for each client. This cost is proportional to the latency between the client and server object and does not depend on the server object's location within the registry system. The comparison focuses on the location and invocation costs exhibited in each model.

Local References

A local reference is defined as a reference located or relocated through a single object. This can be through a local ORB, which manages the registration for the server, a home location or a forwarding object. A reference is local when the server object and client object can communicate within a latency bound of n. Migration for the local reference case is assumed to be within the same latency bound; hence an indirect path between any two objects within the latency bound has a maximum latency bound of $2n$.

In a home location model each client accesses the home location, which then forwards all communications to the mobile object. Given a latency of n to communicate with the home location and then a further latency of n to forward the request to the mobile object. The latency model of location and invocation is $C_{hl} = 2n$.

The forwarding location model for a local reference is defined by a chain of forwarding objects each within at most $2n$ of each other. For each access, where the chain of forwarding pointers has to be traversed in its entirety, the latency is proportional to the length of the chain. For a chain of length t, where the mobile object has relocated $t-1$ times, the average latency model for resolution, and invocation, is $C_{fc_{min}} = tn$. The maximum latency model for resolution in this case is $C_{fc_{max}} = 2tn$. A stub-scion pair [164] (SSP) chain model has the same latency model for resolution, however for invocation the latency is reduced to the latency of a direct reference or forwarding chain of length 1. The latency model for invocation in a SSP chain model is n.

Object location within the distributed ORB system is performed through contact with a local ORB. In the local reference case, this local ORB is assumed to contain information about the required server object. The latency model for location is similar to that of the home location model and can be described as $C_r = 2n$. Invocation occurs through a direct connection to the server object and is hence bounded by n.

Domain References

The home location model has the same cost for the domain case as it does for the local case. That is to say, references must be within n of the home location; it is assumed that the client object resides within the same latency. The cost models for both the forwarding chain and the SSP chain models are also the same since the objects do not exceed a distance of $2n$.

A registry system model exhibits different costs when a mobile object resides within a single registry domain. The minimum cost is incurred when the server object happens to reside at the same ORB that the client contacts; this is equivalent to a local reference. The maximum cost of this model requires an access to the primary registry (with an average latency of n) and access through the local base registry. The cost model for this is $C_{r_{max}} = 4n$.

A registry-based model will outperform a forwarding chain model for a chain length of greater than 4 hops. For the local reference case, the home location model outperforms the registry model in pure latency, however this does not take into account saturation costs due to the home location acting as a communications bottleneck. In Section 8.5.3, the actual costs and saturation levels for these systems are compared and analysed.

Global Reference

When managing global references, or objects that are widely distributed, greater latency costs come into consideration. The home location model incurs greater cost in this case as it not only suffers from saturation problems but also the trombone problem [43, 152].

Figure 64 shows the costs models for the home location, forwarding pointer chain and distributed registry system in the global referencing case. Part (i) of Figure 64 refers to the cost model for the home location case. In this example, the client is latency n away from the home location as assumed in previous models, and the mobile object has relocated to a latency m away from the home location where $m \gg n$. The cost model for the home location model in this case is $C_{hl} = n + m$ where $m \gg n$.

Part (ii) of Figure 64 shows the cost model for a forwarding pointer chain or SSP chain model. In this figure, n is the latency from the client to its first link into the forwarding pointer chain. To resolve the name to a reference, the client must follow the chain of forwarding pointers of length o. The cost model for a forwarding pointer chain in this case is $C_{fp} = on + m$. An SSP chain mechanism will incur this cost for resolution but once resolved, the reference will form an optimised direct reference to a scion based with the mobile object, with a direct latency of $n + m$.

Figure 64, part (iii) shows the cost model for location using the update/ORB-based model. The cost includes the cost to access the primary registry and then further costs as administrative nodes attempt to locate the object. As resolution requests travel further up the ORB hierarchy, the cost of location increases. The location costs for this model have already been examined in Section 6.4 with the costs dependent on the number of hierarchy levels involved and the number of broadcast requests that are required before the reference is resolved. Given a cost of q for communication between administrative nodes, the average cost of location is:

$$C_{init} = 2n + m + \sum_{i=1}^{k} q + 1$$

The cost of invocation is m.

These cost models show that for small systems with infrequent invocations a forwarding location model or a home location model can provide equal if not better performance to an update/ORB-based location model. As the size of the object system increases, it can be seen that these models are effected by both bottleneck issues and poor scalability. The update/ORB-based model supports large scale systems and provides fault resilience through an adaptable registry system and a lack of residual objects.

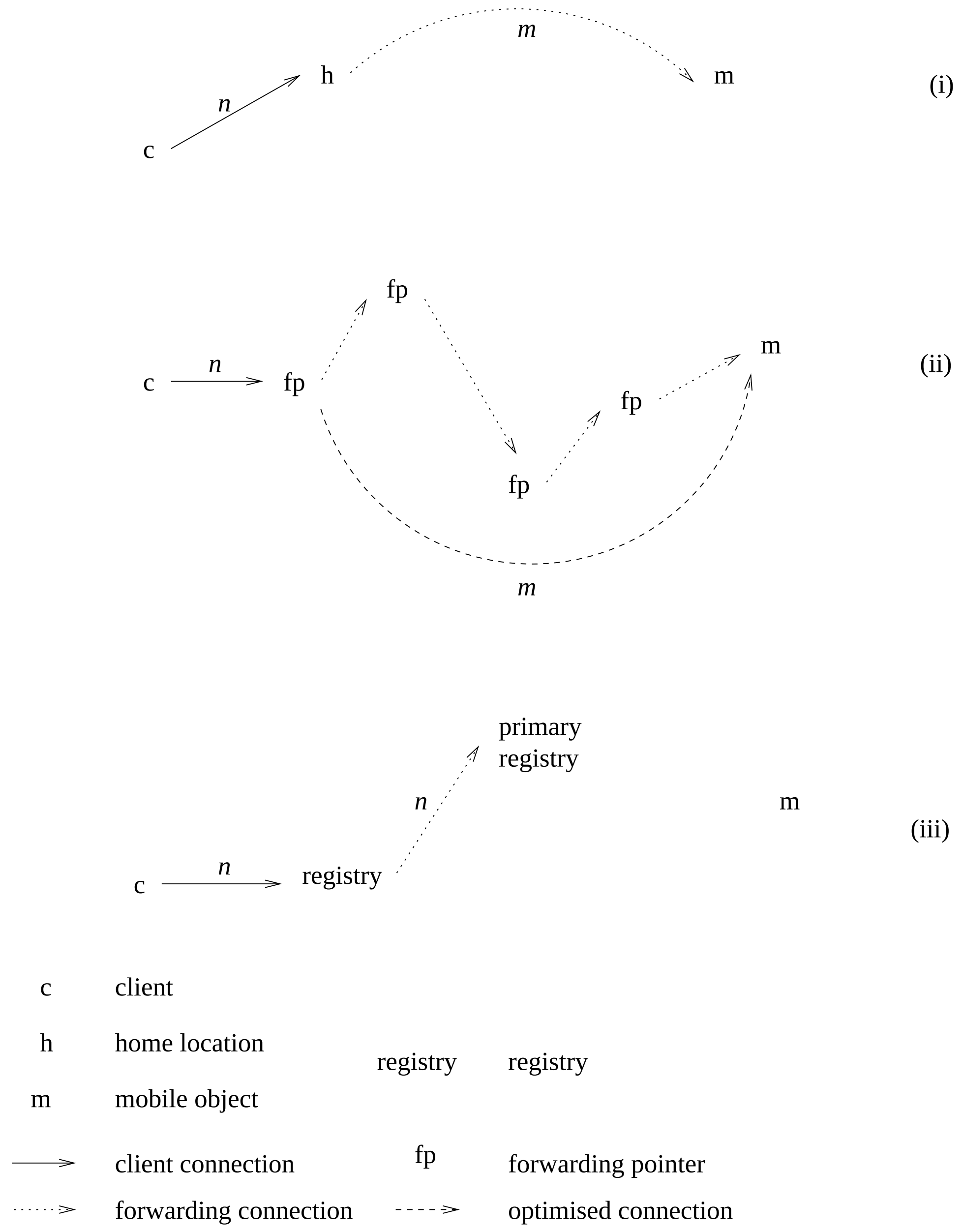


Figure 64: Comparison of global referencing models.

Model	Client Scalability	Frequency Scalability	Distribution Scalability	Server Scalability
Home Location	$\times$	$\times$	$\times$	$\surd$
Forwarding Location	$\surd$	Partial	Partial	$\surd$
SSP	$\surd$	$\surd$	Partial	$\surd$
Central Registry	$\surd$	$\surd$	$\times$	$\times$
Broadcast	$\times$	$\surd$	$\times$	$\surd$
Update/ORB	Partial	$\surd$	$\surd$	$\surd$

Table 13: Scalability of the relocation transparency models.

8.5.2 Scalability

Table 13 shows a comparison of different forms of scalability in the relocation models examined in this thesis. As well as the commonly used home location and forwarding location models, the update model is also compared within the SSP chain, central registry and broadcast models. Each system is classified according to its support for client, access frequency, distribution and server scalability.

Client scalability indicates how well the model supports large numbers of clients; this property is important in systems where large numbers of light weight clients may be connecting at any time. Server scalability indicates how well the model supports large numbers of server or mobile objects. Distribution scalability indicates how well the model copes with the server or mobile objects being distributed over a wide area. Access frequency scalability indicates how well the model copes with frequency of access in terms of frequent client or server access.

Client scalability is provided in the forwarding location models and central registry model as the number of clients may be distributed over multiple machines. This form of scalability is poor in the broadcast and update models as there must be interaction by each client when the server migrates, *i.e.* each client must either broadcast or must receive an update message. The update model reduces the costs of this form of scalability by only requiring update messages for connected clients. The DISCWorld model uses the ORB system to manage its distributed namespace, but provides a WWW-based front end to its services. In the DISCWorld system, client accesses are passed on to the ORB system through several well defined interfaces. In this model, the number of clients is reduced while the frequency of client communication is increased.

Frequency scalability is supported in all systems except for the home location and partially in the forwarding location model. These models do not support frequent client access as they require either transmission through a single point or transmission through multiple objects that must be traversed by each invocation. The other models differ in that they either provide a mechanism for further invocations to be more direct (SSP) or they have a separate channel to the server.

The update model is the only model to fully support distribution scalability through the use of a distributed ORB system that maintains the global namespace, and direct connection between client objects and server objects. This form of scalability is supported partially in the forwarding location models depending on how direct the path is through the migration chain.

Server scalability is the effect of multiple servers or mobile objects within the system. This effect can be seen as either an increased number of broadcasts or replication updates. This model of scalability is supported in all systems except for the central registry where each server requires replication and cached update costs.

8.5.3 Comparison of Models

When evaluating a location and relocation model it is important to contrast the model's performance against that of other commonly used models. Section 8.5 presents models of the performance costs of object location and communication within the update/ORB-based, the home location and the forwarding location models. This section presents a performance analysis of these models and discusses the measurement results with regard to the defined models.

Figure 65 shows the cost of connection for the home location, forwarding location and update models with an increasing number of client requests. These measurements were taken on a farm of AlphaStations with JDK 1.2.2 under low system load; the results shown are the average of 100 measurements for each data point. By increasing the number of concurrent client requests, the scalability of each of these models can be examined. The system conditions for these measurements are a remote client connecting to a local server within the update model and a migration chain length of 2 (a single migration) within the forwarding location model. This provides optimal conditions for the forwarding location model. Connection within the update model is equivalent to the cost of location plus the cost of connecting the client interface fragment with the migration fragment. In this example all locations are through a single, local ORB.

As described in Section 8.5, the cost of connection, with the given conditions, for the home location and forwarding location model is $2n$. The cost of the update model is the cost of location plus the cost of fragment connection. The same protocol, and hence protocol objects, are used to communicate requests in each of the models.

Figure 65 shows that the connection costs for the traditional home location and forwarding location models are significantly higher than those shown by the update model. The multithreaded nature of the ORB system allows the concurrent handling of requests providing greater scalability. Figure 65 also shows the results for a multithreaded home

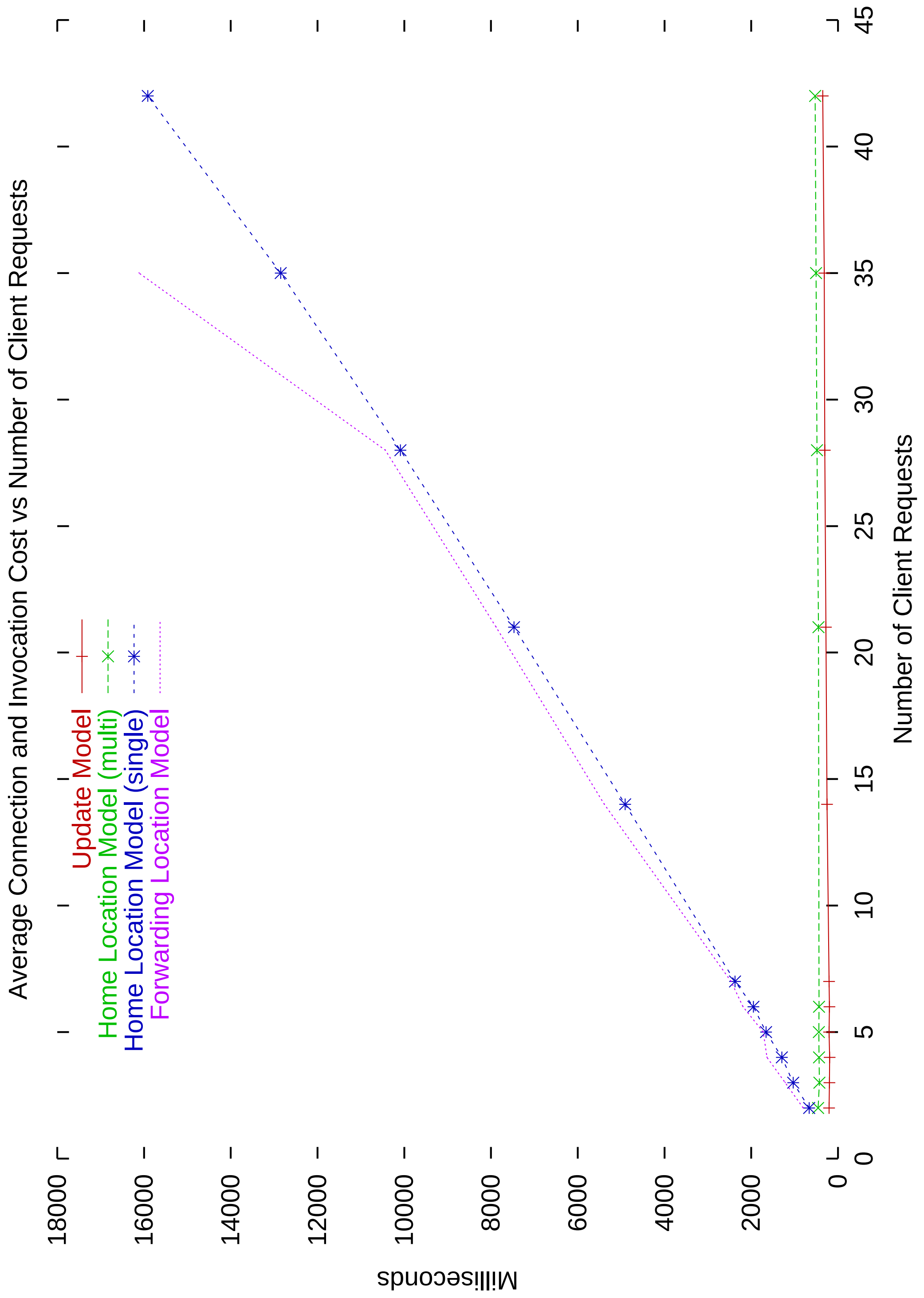


Figure 65: Connection costs in the home and forwarding location, and update models.

location model. The costs for this model are similar to those exhibited by the update model. An optimised forwarding location model is also possible as shown by the SSP chain model. This requires full chain traversal for the first access leading to the same results as the unoptimised version. Further communications exhibit the same results as the optimised update and home location models.

Figure 66 shows the cost of connection for the optimised home location model and two cases of the update model; the two cases considered are where client access is through the one base ORB, and where client access is distributed throughout a registry domain. These measurements, and those of Figure 67 were obtained using the same method as used for Figure 65. This figure highlights the increased cost of connection as the number of client requests within the system increases. The costs for the home location are higher than both cases of the update model, although as client numbers increase the costs of the home location and non-distributed update models converge. These costs are both higher than those exhibited by the distributed update model due to communications bottleneck issues. In the examples of the home location and non-distributed update model, each client connection must be performed through a single home location while the costs within the distributed update model are distributed over the base registries within the registry domain.

Figure 67 shows the connection costs of the distributed update model in detail. The average costs for the non-distributed update model are shown for comparison purposes. This figure shows the connection costs associated with each ORB within the registry domain. In this example there are 7 base registries with *turquoise* as the server object's local ORB. As is expected the cost of access through *turquoise* is always lower than the costs incurred by the other ORB nodes. As can be seen in Figure 67, the scalability of location, with regard to the number of clients requesting location, increases when location requests are distributed over multiple registry nodes.

To provide scalability, as defined in [10], the performance of connection must be of $O(log\ n)$ where n is the number of objects within the system, including client objects. The performance of the update/ORB-based location mechanisms is within the bounds of an $O(log\ n)$ system, for both the non-distributed and distributed location forms.

Figures 65 and 66 illustrate that, as defined in the connection models, the communication cost for the update model is better than that shown by the home location and forwarding location models without requiring residual code objects or suffering from bottleneck issues. By using a distributed ORB system to manage the global namespace, the object system provides a location mechanism for resource discovery as well as distributed access to each server object.

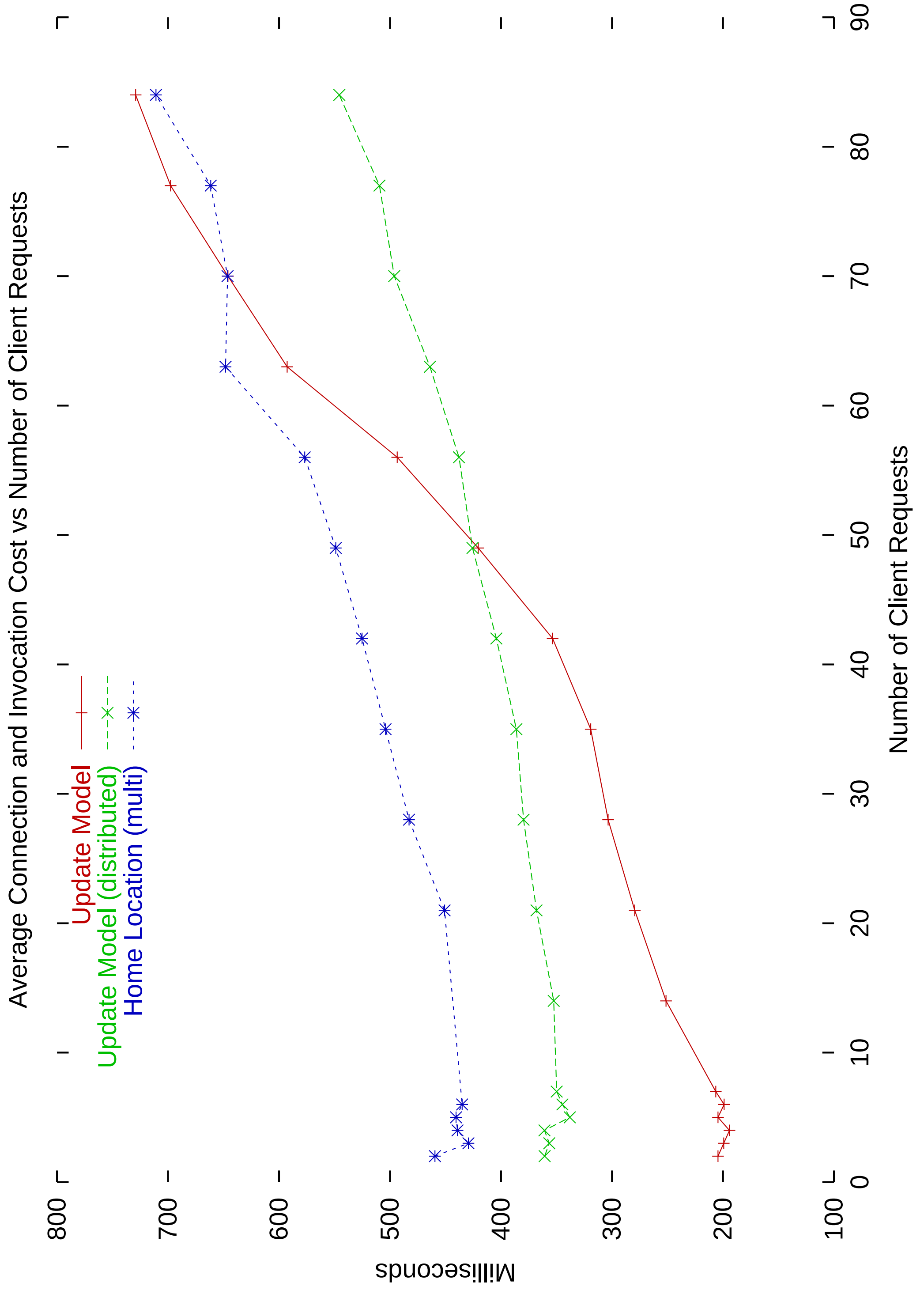


Figure 66: Connection costs for the home location and update models.

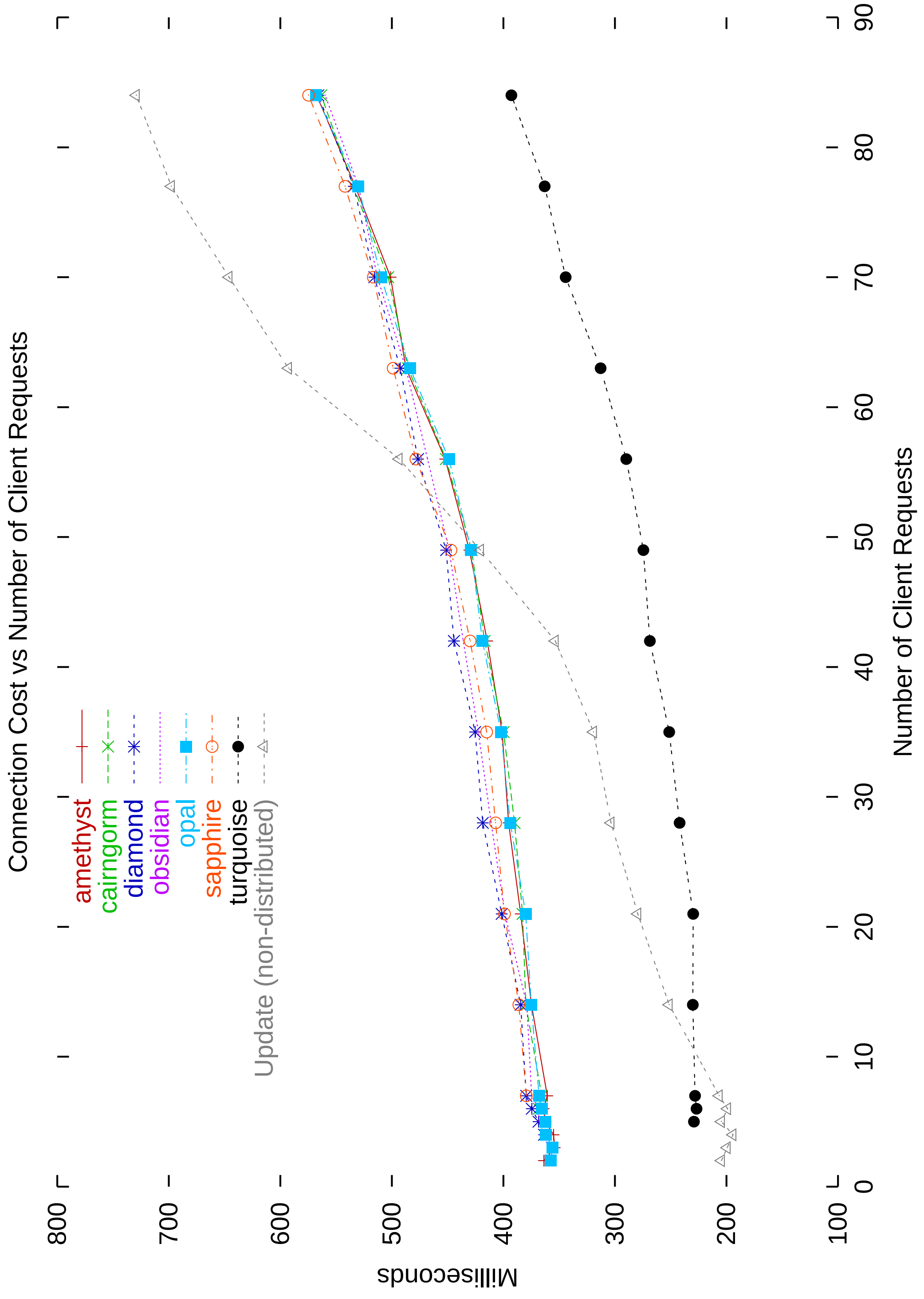


Figure 67: Connection cost and scalability in the distributed update model.

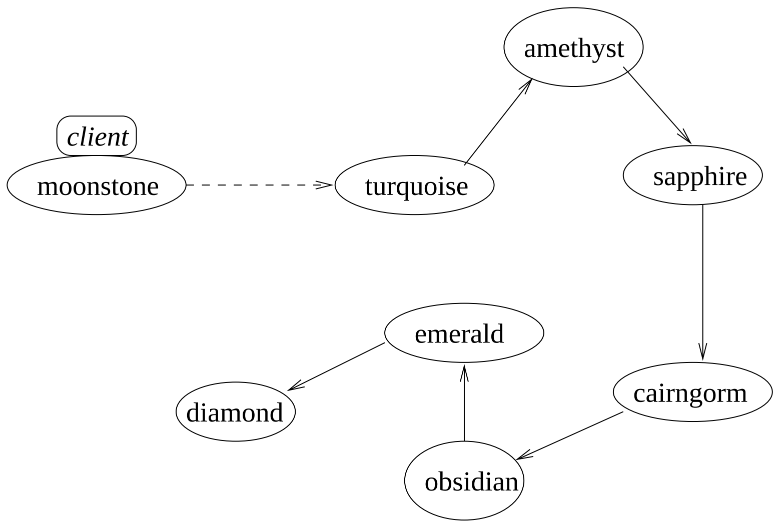


Figure 68: Example migration path used in cost evaluation.

8.5.4 Migration and Relocation Costs

The cost incurred by a client when a mobile object migrates can be divided into three segments: the cost of the method invocation (if an invocation is involved), the cost or delay in migration itself, and the cost of relocation after the migration is completed.

Migration of a server object causes delays for any invocations queued during the migration process. Queued invocations must be stored at the mobile object's previous location and, once migration has completed, be forwarded to the new location for execution. This migration cost effects all migration systems and is not specific to any one form of location or relocation model.

Figure 68 shows an example migration that shall be used as the basis for cost comparison in this section. This figure shows a six stage migration over the AlphaStation farm of machines. In each migration the mobile object started at *turquoise* and ended at *diamond*. The client (in the case of a single client) was resident on *moonstone*. In the home location model, the home location for the object was also resident on *moonstone*, similarly a forwarding object existed on *moonstone* as a starting reference in the forwarding location model. In the update model, a base level registry was resident on each node within the system, with the primary node resident on *sapphire*.

Figure 69 shows the effect of migration on current invocations on a server object. These measurements were performed on a farm of AlphaStations with JDK 1.2.2 under low system load; the results shown are from a single migration run. In this example, a stationary client object is invoking methods on a mobile object at steady intervals corresponding to peaks in the figure. At different stages during the invocation cycle, the mobile object migrates to a new host following the pattern specified in Figure 68. For example, the server object is migrating during invocation number 5. The migration results in additional delays while the

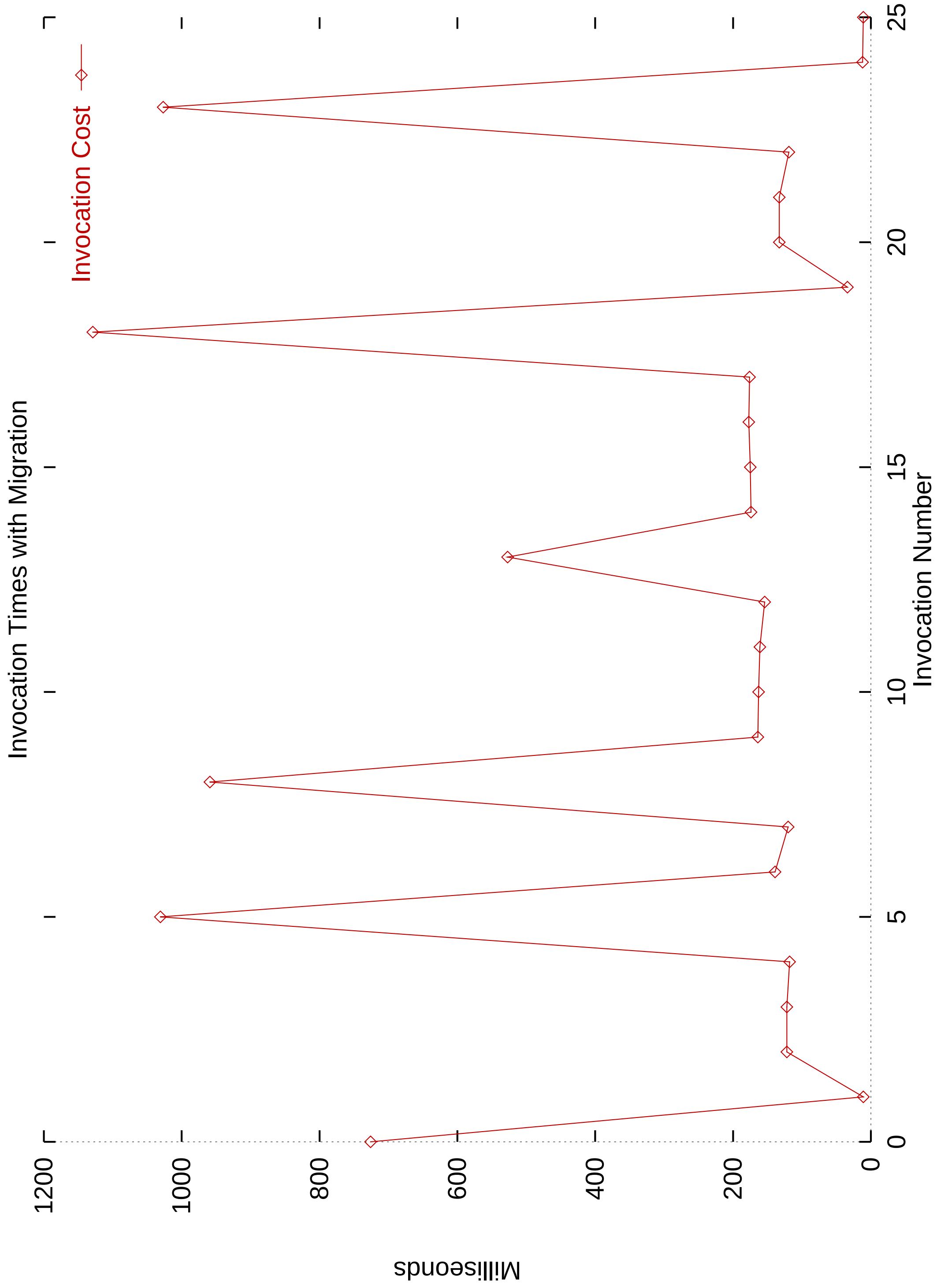


Figure 69: The effect of migration on invocation cost.

invocation is queued, the server object is migrated, and then the invocation is completed at the new location. Additional migrations occur at the invocations numbered 8, 13, 18 and 23. The costs of migration differ between these invocations due to the different times of invocation receipt. An invocation that is queued at the beginning of the migration process will have a longer delay than an invocation that is queued towards the end of the migration process.

The task of fragment reconnection within the update model executes concurrently with invocation forwarding and execution. This enables each client to minimise the cost of relocation and migration. This concurrency can also be implemented in the home location and forwarding location models.

The cost of relocation within the update model consists of the cost incurred while reconnecting the client interface and migration fragments within an invocation. This cost is exhibited in the forwarding location model through an additional link in the migration chain and in the home location model through the cost of reconnecting the home location to the mobile object. The cost of reconnection in the home location model is incurred once for each relocation regardless of the number of clients.

Figure 70 shows the pure relocation cost for the home location, forwarding location and update models. These measurements were taken on a farm of AlphaStations with JDK 1.2.2 under low system load; the measurements shown are the average of 100 measurements for each data point. As expected from the cost models the cost of relocation in the forwarding location model increases at each migration proportional to the size of the migration step. A migration step of approximately 350 milliseconds was incurred. This migration step is the additional cost of forwarding an invocation request protocol object through the new link in the migration chain. Similar results were obtained for the SPARC platform.

The home location model requires a two step relocation or invocation. Each message must be forwarded through the home location to the mobile object, requiring an additional (local) communication between the client and the home location. The cost of relocation is unrelated to the length of the migration chain, as can be seen in Figure 70. The average of the relocation costs within the home location model is 749.71 milliseconds.

The relocation cost for the update model is also unrelated to the length of the migration chain and is a single step relocation. Each relocation cost is related to the reconnection of the interface fragment socket to the new migration fragment socket and the processing of the delayed invocation. The average cost of relocation in the the update model is 507.71 milliseconds. A failed reconnection requires a new location through the ORB system. These costs are detailed in Section 8.4. Failed reconnections in the home location and forwarding location models can not undergo relocation unless additional mechanisms are provided by the system.

The update relocation model is the best of these model examined within the DISCWorld ORB system, based on performance, scalability and fault resilience. The optimised home

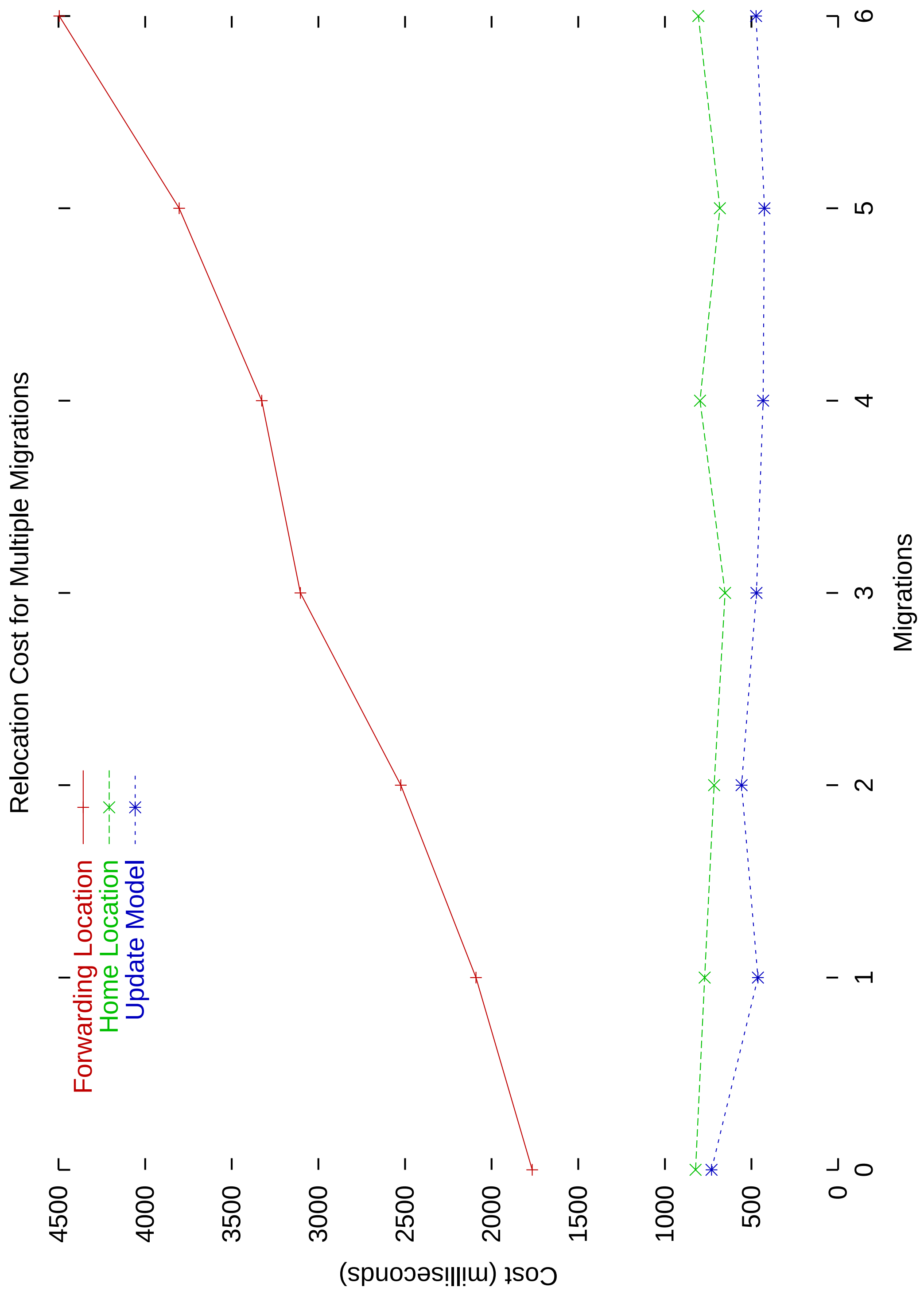


Figure 70: Costs of relocation models.

location model provides almost equal scalability and performance to a single ORB node, but does not provide any fault resilience mechanisms or distribution scalability. The forwarding location models provide client and distribution scalability but not performance scalability. The forwarding location models also suffer from fault resilience problems.

An additional cost of the update model is that an update message must be sent to each connected client. This incurs an additional network communication cost but it is a cost that is distributed over all connected clients. In a similar way to the broadcast model, the update model can lead to network saturation as the number of connected clients becomes large. The same kind of network saturation can occur in the forwarding location model due to vast numbers of communications proceeding through an extensive migration chain, and in a focused manner in the home location model as vast numbers of communications will saturate the direct link between the home location and the mobile object.

In the DISCWorld system, clients access the DISCWorld ORB system through multiple defined WWW-based gateways. These gateways act as an abstraction for client access and reduce the effective number of connected clients within the DISCWorld system. For these reasons the update relocation model is particularly suited to a middleware environment like DISCWorld. The update model is also suited to a system that does not have a large number of clients per server object.

8.5.5 Summary

Location transparency within an object system enables objects within that system to communicate without specifying location details. This transparency requires an abstraction to be made between an object's name and its location; hence a mechanism for obtaining a location from a location transparent name must be provided. The transparent location mechanisms provided by the DISCWorld ORB system rely on a distributed namespace that is able to map abstract names to references that contain cached location information. This mechanism provides a form of location independence.

Relocation transparency within an object system enables objects to communicate throughout object migration without specifying migration details or requiring explicit reconnection. Common models for providing relocation transparency, including the home location and forwarding location models, do not additionally support location transparency.

This section compares commonly used models for providing relocation transparency with the update/ORB-based model used within the DISCWorld ORB system in terms of each models' efficiency and scalability. The DISCWorld ORB system provides better performance than the existing models for object location and relocation costs. The DISCWorld ORB system is scalable with regard to the number of nodes within the system, the number of objects within the system, and the frequency of requests. The distributed nature of the DISCWorld ORB system and the distributed management of the global namespace provides fault resilience. The DISCWorld ORB system provides comparable scalability in terms of the number of client objects with but with additional network traffic costs due to the transmission of multiple update messages. These costs are not directly seen by the client as they are distributed throughout the network.

The results shown in this section indicate that the DISCWorld ORB system is best suited to a distributed object system that requires scalable distribution, fault resilience, and transparent object location and relocation. This system is suited to a system where requests and migration are frequent.

8.6 Summary

The DISCWorld ORB system combines a generic naming model, a relocation transparency model and an ORB-based location model to support location and relocation transparent naming within a distributed and mobile object system. A distributed ORB hierarchy is defined as a dynamic and adaptable structure. Adaptability is used as a fault resilience mechanism that allows the object system to overcome partial node failure and to rebuild itself to operate effectively. The DISCWorld communications protocol is designed in an extensible and flexible manner with the consequence of additional overhead in protocol evaluation. The development of the DISCWorld ORB system and the DISCWorld communications protocol in Java has also incurred additional performance problems.

The global namespace is distributed throughout the ORB system as database segments. There is support for all attributes and preferences defined within the extended naming model. Some of these attributes require more expensive implementation support than others due to additional costs in maintaining global namespace coherency. The default naming model represents an efficient and flexible naming model.

The cost of object location within the DISCWorld ORB system is measured for objects within a location ORB, a registry domain and within a level 2 hierarchy. These measurements show that the cost of location is slightly dependent on the size of the namespace and the position of the required name within the namespace. A single node database segment can hold up to 1000 names without causing performance degradation.

The performance of the DISCWorld ORB location and relocation model is compared to the home and forwarding location models. This comparison shows that the DISCWorld ORB model provides better performance with regard to object location and relocation costs. The scalability of the DISCWorld ORB system is also compared against that of other, commonly used, models of relocation transparency. These models provide relocation transparency but no form of location mechanism or failure resilience. The DISCWorld ORB system is scalable with regard to the size of the global namespace, the number of client requests within the system and the distribution of the namespace. The DISCWorld ORB system provides distribution scalability and an adaptive response to node failure.

Chapter 9

Summary and Conclusions

Relocation transparency within a mobile object system enables mobile objects to migrate while transparently maintaining communications with potentially remote objects. Several models to support relocation transparency have been suggested and used in existing mobile object systems. These models exhibit common failure points and a lack of scalability. This thesis proposes a unique combination of an update model and an ORB-based naming system to provide both location and relocation transparency in a mobile object system. To support widely distributed object and client/server systems the namespace for this system is distributed throughout a hierarchy of ORB nodes.

The naming model used within a distributed or mobile object system defines semantic properties and restrictions on the performance and efficiency of the system. Existing classification models are examined and are found to be unsuitable for the classification of existing mobile and distributed object systems. This thesis proposes a model for classifying naming systems as an extension to previous work in the area. This extension is then used to construct a generic naming system that allows the dynamic definition and adaption of naming models within an object systems.

A distributed object system that supports both the proposed model of relocation transparency and the generic naming model is developed. The effects of the naming model and the transparent relocation model are discussed and analysed both qualitatively and through performance measurement. This distributed object system is implemented as an ORB system that supports scalable and transparent object location, relocation and migration.

9.1 The Need for Transparency and Naming Models

Location transparency within an object system removes visible location information from object references and removes the need to specify the location of an object when communicating with that object. A *name* is often used as a reference to an object; it is this name that must exhibit location and relocation transparency. For a name to be also location independent it must not contain any visible or invisible location information. Location independence is not a requirement of relocation transparency, however, Saltzer [158] states that a reference must be location independent to remove contextual limitations within the system.

Relocation transparency relies on the ability to abstract or mask object migration from client references. Many different mechanisms have been used in mobile and distributed object systems to date, including the home location model [12, 34, 49], forwarding location model [4, 6, 61, 84, 116] and variations [164], broadcast [124] and centralised and distributed name server systems [29, 131, 154]. Many of these systems base their simplicity and abilities on the dependence and/or opaqueness of their references. These systems also have several performance problems and failure points as described in Chapter 1.

9.1.1 The Update Relocation Model

This thesis proposes a transparent relocation model that combines an explicit update model with an ORB-based location system. A global namespace distributed throughout an ORB hierarchy is used to support object location and relocation in case of update failure. Names are managed according to a formally defined but unspecified naming model.

References within the update/ORB-based model are location and relocation transparent. References are partially location independent; references can contain some location information but this information is not required. These references are managed using a fragmented object model. A client interface fragment is used as a reference and a migration fragment handles communication receipt at the server end.

The update/ORB-based mechanism considers a three stage lifecycle for references. The first stage is of an unknown references. Unknown references are names that have not been resolved to a reference and hence can not contain any location information. The second stage is of an unconnected reference. Unconnected references have been resolved and can contain some location information. The third stage is of a connected reference. A connected reference contains location information which is validated through communication with the server object at the location.

Object location corresponds to the stage of transforming an unknown reference to an unconnected reference. Object relocation occurs when the server object migrates and the information contained within the connected reference becomes out-of-date.

The methods commonly used in existing mobile and distributed object systems require the creation of residual code objects. These residual objects contain cached location information. By contacting one of these residual objects, either a home location or a forwarding location, a client is able to, at some point, contact the server object. The failure of a residual object causes a failure in the location/relocation mechanism. Some systems use a backup broadcast or centralised registry system to overcome these failure points. These backup models are expensive and are not able to provide facilities such as resource discovery. The update/ORB-based relocation model does not require the creation of residual code objects and provides resource discovery and attribute-based location mechanisms.

9.1.2 Formal Naming Models

The binding of names has been studied in depth by Saltzer [158] and Bayerdorffer [15, 16] and has been examined with respect to concurrent object systems [15, 16], addressing architectures [50, 63, 92, 158, 174] and resolution systems [26, 174]. A detailed study on naming within mobile and distributed systems does not exist.

Bayerdorffer's work examines and presents a formal model for the classification of name binding systems within concurrent object systems. Name binding is the act of matching a name with an object. Bowman *et al* [26] defines a formal name resolution model for use within attribute-based resolution systems. Name resolution is the act of determining the bound object from a given name. Attribute-based resolution determines the bound object from some, often semantic-based, description.

This thesis presents a classification of existing mobile and distributed object systems with regard to the naming models developed by Bayerdorffer and Bowman *et al.* The process of this classification shows that these models are not suitable for a complete and accurate classification of these kinds of object systems. Additionally no classification of the location independence or transparency of the system is provided. An analysis of these example classifications and the requirement to include appropriate classifications for location independence and location/relocation transparency motivates the development of a new naming model based on this previous work.

The model presented by Bowman *et al* is defined formally. This definition promotes flexibility and produces a mechanism for the generic specification of a name resolution model. This mechanism is applied to the work by Bayerdorffer and the extended naming model developed in Chapter 5 to produce a formal system for the validation of name bindings against a generic naming model.

9.1.3 The Distributed ORB System

The DISCWorld ORB [51, 104] system is a distributed ORB system designed to support object mobility and replication. Location and relocation transparency are supported through the update relocation model and through a global namespace managed by the ORB system. The DISCWorld ORB system operates within the DISCWorld metacomputing environment [78, 82, 104].

The formal model for naming system classification is used by the DISCWorld ORB system to define support for a generic naming model. These naming models can be specified on a single ORB or ORB group basis. The generic naming model provides the support required for distributed resource location and name management. The formal naming model is designed to be logically separated from the remainder of the naming or ORB system. This separation provides support for further extension and additions to the naming model and the ability to change the naming model for a given object system.

ORBs within the DISCWorld ORB system are organised into a structured hierarchy containing active and administrative nodes. The global namespace supported by the naming model is distributed throughout this hierarchy. Active nodes are responsible for managing segments of this global namespace, while administrative nodes are responsible for gathering and spreading the information contained within the segments in response to remote requests for resolution and binding. ORBs are able to adapt their position within the hierarchy in order to minimise the effect of node partial failure. This minimisation enables the system to exhibit gradual performance decay rather than complete subsystem failure. An adapted ORB structure is able to regain all information lost through partial node failure and eventually achieve full performance.

A specific naming system classifiable by the formal naming model is defined as a default naming system for the DISCWorld ORB system. This naming system is defined to support reference transparency and independence.

Each ORB is responsible for managing a segment of the global namespace. ORBs are also responsible for providing additional services such as migration and replication support services. These services are accessed using defined application programming interfaces. These interfaces define the set of operations that are available to client and server objects within the system, and act as an abstraction to the underlying communications protocol.

By using the formal naming classification models and the relocation model described in this thesis, the DISCWorld ORB system is capable of supporting location and relocation transparency on a wide-scale with frequent object migration and object invocation.

9.2 Contributions and Final Conclusions

There are three main contributions of this thesis. The first contribution is the development and analysis of a formal naming model for classifying the naming systems of distributed and mobile object systems. The ability to correctly classify and define naming systems according to some naming model is an important component in system development. The proposed naming model is particularly suited to the classification of object systems with requirements for mobility and location independence or transparency but is capable of being used to classify distributed and mobile object systems with a subset of these characteristics.

Given a formal model for a naming system an object system can be developed with a separation of naming model from naming system and hence a separation of naming system from the remainder of the object system and infrastructure. An implementation of such a system is capable of supporting any naming model that can be defined by the formal classification model. A flexible and extensible naming model can be used to represent a system with varying levels of transparency and independence.

The second contribution of this thesis is the development of a model for location and relocation transparency within widely distributed mobile object systems. This model uses an update model and a distributed namespace to provide these forms of transparency. The update model used to provide relocation transparency avoids the central points of failure and residual object problems found in other commonly used transparent relocation models. The model of location transparency additionally provides support for resource discovery and attribute-based name resolution.

The third contribution of this thesis is the implementation of a distributed ORB system designed to manage a distributed global namespace and to provide service support for object naming, location and migration. The DISCWorld ORB system's transparent relocation model provides significantly better performance than other commonly used models. The ORB system is hierarchically structured, providing namespace scalability and fault resilience through adaption. Communication mechanisms designed to maximise remote communications efficiency are introduced; these communications mechanisms require the relocation transparent support provided by the ORB system and the update relocation model.

The global namespace is managed through the formal definition of a generic naming system. A naming system is specified by the naming characteristics or attributes that it must model, and is used to provide name binding, management and resolution support throughout the ORB system. The choice of naming system has great effect on the efficiency and behaviour of an object system. The choice of an optimal naming system for the development of a distributed ORB system designed to support transparent location and relocation can be performed through analysis of commonly used naming systems and the attributes that define a naming system.

Within the DISCWorld ORB system, relocation transparency is provided for mobile objects without the need for residual code or full use of location dependent information. Transparent location and relocation of objects within the DISCWorld ORB system can be performed efficiently through the use of an update relocation model and a distributed namespace. The hierarchical structure of the DISCWorld ORB system enables the object system to support scalable and efficient location in a distributed global namespace. This scalability extends to scalability in terms of the size of namespace managed in a registry domain, the number of nodes within the system and the frequency of client requests for object location. This combination of efficient object location and relocation results in an object system that can be used to support frequent object migration and remote communication in a widely distributed object system.

The DISCWorld ORB relocation model provides better performance than commonly used relocation models in terms of object location and relocation costs. The DISCWorld ORB system is also more scalable than these models, providing additional location and failure resilience mechanisms. The DISCWorld ORB system is scalable with regard to the size of the global namespace, the number of client requests within the system and the distribution of the namespace. The DISCWorld ORB system provides distribution scalability and an adaptive response to node failure.

The classification and relocation models presented in this thesis provide benefits outside of the DISCWorld metacomputing environment. The relocation model (and its associated location mechanisms) can be implemented in any ORB-based system or any system with a sophisticated name server. The classification model for naming systems can be applied to naming systems within mobile object, mobile process, mobile computer and distributed systems as has been shown in this thesis.

9.3 Future Work

The work described in this thesis consists of the development and analysis of naming and relocation models used within mobile object systems with a requirement for transparency. An implementation of a distributed ORB system to support these models in an efficient manner is described in its prototype form. Future directions for this work are numerous and consist of extensions to the work discussed and optimisations for the implementation system.

The distributed nature of the namespace supported by the DISCWorld ORB system means that some of the naming model's supported attributes are expensive to implement. These attributes require that the entire database be examined to validate bindings and name management. A form of contextual naming could be a solution to this problem. By using each registry domain as a context and requiring unique naming (dependent on the naming model) within these contexts the expense of verification can be reduced. However, the benefits of global naming are also reduced. An extension to this mechanism could be to propose a new category of aliasing apart from the existing local and global categories. This new category, a domain category, can be used to provide contextual naming within a single registry domain.

The extended naming model has been applied as a classification tool to several existing mobile and distributed object systems. Attention is focused on object systems that have some support for wide-scale client/server computing, location independence and transparency, relocation transparency, interesting naming systems or mobility. Further classification of object systems outside of this domain is an interesting future consideration (particularly a reclassification of the concurrent systems classified by Bayerdorffer).

The transparent relocation model described in this thesis is used to support transparent object relocation in a client/server object system. The update model is shown as a competitive model compared to other commonly used relocation models while removing some common points of performance degradation and failure. The relocation model used utilises a combination of two relocation models. Consideration of further combinations, such as those described in MOA [131], could be used to highlight further combinations that increase either quality or performance of transparent object relocation.

The implementation of the prototype DISCWorld ORB system requires and provides room for several optimisations. All of the components of the DISCWorld ORB system are implemented using Java [69] which, as an interpreted language, has been shown to have performance problems [111]. The problems identified in this thesis are associated with Java's serialisation mechanisms [95,119]. Discussion in Chapters 7 and 8 shows the effect of some of these problems, specifically in the areas of protocol object and information transfer between hosts using the provided serialisation mechanisms.

9.4 Finale

By providing location and relocation transparency in an object system, references to objects can be independent of static location restrictions. Within a widely distributed object system, it is important to provide referencing or naming systems that are also distributed and, accordingly, scalable and efficient. The analysis of naming systems performed in this thesis has provided insights into the naming requirements of such a system and has enabled the development of classification models suitable for these types of systems. It is hoped that the classification of desirable characteristics of naming systems will enable the future development of naming models and systems suitable for the growing area of object systems with transparency requirements.

Appendix A

Protocol Specification

The DISCWorld ORB communications protocol is based on the construction of self-evaluating protocol objects. Protocol objects are used to request services and provide information to components within the object system. The DISCWorld ORB communications protocol is designed to support flexibility and extensibility, hence, all protocol objects extend standard interfaces.

Most communications performed by protocol objects are asynchronous and are used to inform system components of changes to the name binding database. Protocol objects used in communication between client and server objects and the ORB system perform synchronous communication.

A.1 Protocol Objects

All protocol objects within the DISCWorld ORB system extend a `ProcessRequestObject`. Each `ProcessRequestObject` contains information concerning its source and destination objects, and the name or object that it is interested in.

Each protocol object contains a method which operates directly on the database contained within a registry. An abstraction over the behaviour of each protocol object can be ensured by each protocol object executing its own requests directly on the database. For example, a protocol object that wishes to register a service can perform this task itself by invoking the appropriate methods on the database. In this way, each registry does not need to have complete knowledge about the protocol objects in existence and the behaviour they require.

The class definition of the `ProcessRequestObject` class is shown below.

```
public abstract class ProcessRequestObject
extends java.lang.Object implements RequestObject

  int getAdapterPort();
  // Returns the adapter port associated with the server.
```

```
void setAdapterPort(int aP);
// Sets the adapter port associated with the server.

java.net.InetAddress getHost();
// Gets the request host.

void setHost(java.net.InetAddress h);
// Sets the request host.

java.lang.String getServerClassName();
// Returns the class name of the Server that this request refers to.

java.lang.String getServerName();
// Returns the name or id of the Server this request pertains to.

void setServerName(java.lang.String sN);
// Sets the name or id of the Server that this request refers to.

void setServerClassName(java.lang.String sCN);
// Sets the class name of the Server that this request refers to.

abstract void process(RegistryData ds,
                      java.io.ObjectOutputStream out);
// Abstract method as a placeholder for subclasses to process
// themselves using the appropriate Registry data structures.

void readExternal(java.io.ObjectInput in);
// Implements externalisation (reading).

void writeExternal(java.io.ObjectOutput out);
// Implements externalisation (writing).

boolean equals(java.lang.Object obj);
// Returns true if two ProcessRequestObjects are equal.
```

Subclasses of this class must implement the process method, performing appropriate operations on the registry data structure and optionally constructing a `ResponseObject` to respond to the request with appropriate feedback. Each protocol object may extend the methods provided through the `ProcessRequestObject` interface to provide methods to assist in its creation.

Each registry is capable of processing protocol objects by invoking its `process` method, passing it the database on which it is to operate.

A `RegisterRequest` is an example of a commonly used protocol object; a `RegisterRequest` is used to register a service, potentially with some policy, with a registry. A `RegisterRequest` extends the `ProcessRequestObject` interface, implementing the `process` method and adding methods to register policies and information on the registered service (such as protocol choices and aliasing). The `RegisterRequest` interface is shown below.

```
RegisterRequest();
// Creates an empty RegisterRequest.

RegisterRequest(java.lang.Object service,
                int aP);
// Creates a RegisterRequest for the service object with the
// specified adapter port.

RegisterRequest(java.lang.Object service,
                java.lang.String sN,
                int aP);
// Creates a RegisterRequest for the service object with a id and a
// specified adapter port.

boolean getGenerate();
// Returns true if the registration requires stub code generation.

Policy getPolicy();
// Returns the policy object associated with this registration.

void setPolicy(Policy pD);
// Sets the policy object.

int getProblem();
// Returns the number of problem reports lodged against this registration.

void setProblem();
// Increments the problem counter.

Support getReplicant();
// Returns the replicator support object for this registration.

void setReplicants(Support rD);
// Sets the replicator support object.

java.lang.Class getServerClass();
// Returns the Class object for the server.

void setServerClass(java.lang.Class sC);
// Set the Class object for the server.

java.lang.Class getStubClass();
// Returns the Class object for the stub.

void setStubClass(java.lang.Class sC);
// Set the Class object of the stub.

void move(MobileInstigationRequest service);
// Moves the server according to the details in the request.

void suspend();
// Suspends operations on this server;
```

```
void process(RegistryData ds, java.io.ObjectOutputStream out);
// Process the RegisterRequest with the appropriate registry data
// structure.

void readExternal(java.io.ObjectInput in);
// Implements externalisation (reading).

void writeExternal(java.io.ObjectOutput out);
// Implements externalisation (writing).
```

`RegisterRequest` objects are stored within the registry database and can be accessed at any point by the registry to peform policy checks and to maintain problem counters[11]. Register requests are responsible for registering the service with the database and for instigating policy checking and evaluation. If registration completes, a `ResponseObject` is returned with a positive value, otherwise a negative value is returned.

The body of the `process` method for a `RegisterReqeust` is shown below.

```
public void process(RegistryData ds,
                    ObjectOutputStream out)
    throws ResponseCorruptionException {

    ResponseObject obj = new ResponseObject(serverName,serverClassName,
                                        this.getClass().getName());
    try {
        ds.add(this);
        obj = new NewPort(obj,this.getAdapterPort());
        obj.setAchieve();
    }
    catch (DuplicateRegisterException e) {
        obj.setAchieve(e);
    }
    catch (PolicyException e) {
        obj.setAchieve(e);
    }
    try {
        out.writeObject(obj);
    }
    catch (IOException e) {
        throw new ResponseCorruptionException(e);
    }
}
```

The action of the `process` method is to attempt to add the service to the database, set the adapter port of the object to be the port of the created adapter and, if these commands were successful, set a positive achievement value. If exceptions are raised, a negative achievement is set and the exception is passed back to the source of the request.

[11]As problems are encountered with references to server objects (dropped connections, high latency) problem reports are sent to the registry which maintains the reference for the object. After a specified number of problem reports, the registry stops handing out references to that object and raises an exception within the remote object if possible.

Any protocol object can operate successfully on a registry database by accessing its common interface methods (*c.f.* Appendix B). A `RegisterRequest` can operate successfully on any registry database that extends the API.

The protocol objects defined for use within the DISCWorld registry system are shown in Table 14. These protocol objects provide the core for internal communications within the DISCWorld ORB system. This can be extended through the development of new protocol objects conforming to the `ProcessRequestObject` interface.

Protocol Object	Usage
RegisterRequest	Registers a service with the registry.
DeregisterRequest	Removes a service entry from the registry.
LocateRequest	Searches a registry database for a name that matches the attributes contained within the request.
QueryAllRequest	Lists all registered services at the base registry.
InvokeRequest	Used to invoke a method (defined by information contained within the request) on the recipient remote object.
ResponseObject	Response to a request that carries with it success information and any results.
FinalRequest	Closes a continuing communications channel.
MobileInstigationRequest	Used to initiate a mobility request causing suspension of request invocations.
MobileRequest	Performs relocation of a mobile object.
MoveRequest	Low level request object that performs movement of object.
SuspendRequest	Low level request object that performs suspension of object.
ProblemReport	Informs registry of problems in communications channels with a remote object.
UpdateRequest	Informs connected client of an update in location.

Table 14: Protocol objects used within the DISCWorld ORB system.

A.2 I/O Automata Definitions

To explore how these protocol objects are used throughout the DISCWorld ORB system, a transition definition is given using a form of Lynch and Tuttle's I/O Automata [118]. I/O automata have been used to describe actions and transitions in systems and to define safety and liveness properties for protocols used in distributed object systems [43] and distributed shared memory systems [53].

The I/O automata is a labelled transition model for synchronous and asynchronous systems (although it is naturally asynchronous [118]). An I/O automaton can be used to define the actions and transitions of a distributed object system through its protocol object interactions. An I/O automaton consists of a set of *states*, a set of *actions*, a set of *steps* and a set of *tasks*; a set of actions can be further categorised into *input*, *output* and *internal* action sets. A step is a transition between two states due to some action and forms a triple: (*state*, *action*, *state*). A task consists of a subset of actions.

To define an I/O automaton for a system, the states and actions have to be defined. The set of states can be defined by the introduction of state variables; the combination of all possible settings of these variables (that arise through all possible actions) is the set of states of the automaton. The set of steps can be defined as the transition relation of the automaton: it is the set of all possible state transitions occuring from defined actions. Each step consists of a guarded sequence of statements and is composed of a *precondition* and an *effect*.

A transition relation of the DISCWorld ORB communications protocol can be developed by mapping the protocol objects sent between client objects, server objects and ORBs with the object system. These mappings produce automaton definitions for which actions can be defined.

A.2.1 Client I/O Automaton

The client has a restricted set of protocol objects that it can accept or send. The action signature, $acts(S)$, (set of actions) for a client consists of:

Input Actions:	ResponseObject
	UpdateRequest
Output Actions:	LocateRequest
	QueryAllRequest
	ProblemReport
Internal Actions:	Connect
	ConnectionFailure

The state of the client's I/O automaton consists of the boolean state variables *waiting* and *connected*, the integer variable *problemcounter* and a cache of location hints *cache*. For simplicity, it is assumed that the client has only one server reference at a time. The case of several connections does not change the effect of the transition relation but complicates it.

The variable *waiting* is true if the client is waiting for a response object (it may be waiting for multiple response objects at once and may be continuing execution), and false otherwise. The variable *connected* is true if the client is connected to the server, and false otherwise. The integer *problemcounter* keeps a record of the number of problems registered against the server.

Given these variables and the defined action signature, the transition relation can be defined as:

```
ResponseObject
    Precondition: waiting = true
    Effect:       if ResponseObject instanceof reference
                         connected ← true
                  else if ResponseObject instanceof List(reference)
                         return ResponseObject
                  waiting ← false

UpdateRequest
    Precondition: connected = true
    Effect:       cache ← cache + UpdateRequest
                  Connect

LocateRequest
    Precondition: true
    Effect:       waiting ← true

QueryAllRequest
    Precondition: true
    Effect:       waiting ← true

ProblemReport
    Precondition: true
    Effect:       problemcounter ← problemcounter +1

Connect
    Precondition: cache ≠ ∅
    Effect:       $connect(cache)$
                  connected ← true

ConnectionFailure
    Precondition: connected = true
    Effect:       if problemcounter < $n$
                         ProblemReport
                         Connect
                  else
                         ProblemReport
                         connected ← false
```

A.2.2 Server I/O Automaton

The server object also has a restricted set of protocol objects that it can accept or send. The action signature for the server object is:

Input Actions:	InvokeRequest
	MoveRequest
	SuspendRequest
Output Actions:	UpdateRequest
	RegisterRequest
	DeregisterRequest
Internal Actions:	Move

The state of the server's I/O automaton consists of the boolean state variables *suspended* and *finished*, a string state variable, *location*, defining the object's new location, and a queue of incoming invocations, *queue*, that must be performed after the move. To simplify the automaton, it is assumed that each server object has only one client connection at a time.

The variable *suspended* is true if the object is ready to move (it is not currently performing invoke requests but is queueing them instead) and false otherwise. The variable *finished* is true if the object has completed any pending invocations and is ready to move. The variable *location* contains the new location of a move request.

Given these variables and the defined action signature, the transition relation can be defined as:

```
InvokeRequest
    Precondition: true
    Effect:       if suspended = true
                      queue ← queue + InvokeRequest
                  else
                      $invoke(InvokeRequest)$
                      if suspended = true then finished = true

MoveRequest
    Precondition: suspended = true, finished = true
    Effect:       Move

SuspendRequest
    Precondition: true
    Effect:       suspended ← true

UpdateRequest
    Precondition: suspended = true
    Effect:       $nil$
```

RegisterRequest

Precondition:	true
Effect:	suspended = false

DeregisterRequest

Precondition:	true
Effect:	suspended = true

Move

Precondition:	suspended = true, finished = true
Effect:	UpdateRequest
	DeregisterRequest
	move(*location*)
	RegisterRequest

A.2.3 ORB I/O Automaton

An ORB accepts incoming connections from both clients and servers concurrently and hence has an action signature to complement both automata. For simplicity it is assumed that the ORB has only one connection to a client and one connection to a server existing at the one time. Multiple connections do not alter the behaviour of the automaton, however, they do require arrays of server and client objects to be maintained which complicates the automaton.

Base level ORBs handle communications differently to primary level ORBs. The base level ORB shall be considered first and has the action signature of:

Input Actions:	LocateRequest
	QueryAllRequest
	ProblemReport
	RegisterRequest
	DeregisterRequest
Output Actions:	ResponseObject
Internal Actions:	none

The state of the base ORB's automaton consists of the database, *db*, which maintains a server entry with the assumption that there exists a single server entry at a time. The database has internal operations: *add* and *remove*, which allow the database to be altered by the ORB. The base level ORB differs from the primary ORB in that if a query is unsuccessful it forwards the request to the primary and it also forwards all RegisterRequest actions. This forwarding is performed by an internal function, *forward*.

```
LocateRequest
    Precondition: true
    Effect:       if db = LocateRequest
                          ResponseObject(db)
                  else
                          ResponseObject($forward$)

QueryAllRequest
    Precondition: true
    Effect:       ResponseObject(db)

ProblemReport
    Precondition: true
    Effect:       if db = ProblemReport
                          $db.remove$
                  else
                          ResponseObject($error$)

RegisterRequest
    Precondition: true
    Effect:       $db.add$
                  $forward$

DeregisterRequest
    Precondition: true
    Effect:       if db = DeregisterRequest
                          $db.remove$
                  else
                          ResponseObject($error$)

ResponseObject
    Precondition: true
    Effect:       $nil$
```

A primary ORB accepts a subset of the actions accepted by a base ORB, performing only addition, location and removal operations. The action signature of a primary ORB consists of the following actions.

Input Actions:	LocateRequest
	RegisterRequest
	DeregisterRequest
Output Actions:	ResponseObject
Internal Actions:	none

The state of a primary ORB's automaton consists of a database, *db*, which maintains the server entries for all databases segments within the registry domain. For simplicity it is assumed that the primary ORB database only contains one server entry at a time. The database has internal functions: *add* and *remove* which allow the primary ORB to add and remove server entries.

```
LocateRequest
    Precondition: true
    Effect:       if db = LocateRequest
                          ResponseObject(db)
                  else
                          ResponseObject(error)

RegisterRequest
    Precondition: true
    Effect:       db.add

DeregisterRequest
    Precondition: true
    Effect:       if db = DeregisterRequest
                          db.remove
                  else
                          ResponseObject(error)

ResponseObject
    Precondition: true
    Effect:       nil
```

The composition of these automata can be said to be *strongly compatible* [118] for all automata within the composition defined by the action signatures $\{S_i\}_{i \in I}$, if for all $i, j \in I$ where $i \neq j$:

1. $out(S_i) \cap out(S_j) = \emptyset$
2. $internal(S_i) \cap internal(S_j) = \emptyset$
3. no action is contained in infinitely many sets, $actions(S_i)$

This is true for composition of the ORB, server and client object automata as for the out action sets:

$$\begin{aligned} out_c &: LR, QAR, PR \\ out_s &: UR, RR, DR \\ out_r &: RO \end{aligned}$$

and the internal action sets:

$$\begin{aligned} internal_c &: C, CF \\ internal_s &: M \\ internal_r &: \emptyset \end{aligned}$$

as the composition of the out action sets contains no common subsets, the composition of the internal action sets contains no common subsets and no action is contained in infinitely may sets.

An automaton, A, that is strongly compatible can be said to be *fair* if there exists a fair execution of A. Lynch [118] provides the following definition for a fair execution.

Definition A.1 (Fair Execution). A fair execution of an automaton A is defined to be an execution α of A such that the following conditions hold for each class C of $part(A)$:

1. If α is finite, then no action of C is enabled in the final state of α.

2. If α is infinite, then either α contains infinitely many events from C, or α contains infinitely many occurences of states in which no action of C is enabled.

This means that a fair execution of a system is one that gives turns to each component of the system and does not block in a state that does not release control. Lynch has shown that the fairness (and hence liveness) of a system is dependent on the fairness of each of its parts [118].

The automata of the client, server and ORB system is said to be fair if for all valid connections, a connection will be maintained and for invalid connections the connection will be removed. In can be seen in the client's automata that a connection will exist while the server connection is established; if repeated failure occurs, *problemcounter* is guaranteed to increase until it exceeds the allowable failures within the system. The registry will then only resolve names to the object once it has reregistered. Hence the DISCWorld ORB communications protocol provides a fair execution.

Appendix B

API for DISCWorld ORB System

B.1 Registry Interface

The main communication interface for the registry service is provided through the `Registry` interface. This interface acts as the standard communications interface for client and server objects and provides facilities to query the database and add bindings to the database. The provided methods map onto commonly used methods in existing ORB systems (CORBA, RMI) and provide equivalent functionality.

Clients can obtain information about a service registered with the local ORB node, and within the ORB system, through the lookup and query methods. The lookup methods provide a means to search for a particular service while the query methods provide a way for clients to find out what services are provided at a host and what choice of service they have within a service group.

Servers are able to register a service through the register commands. Depending on the type of service required, different register methods will be invoked, taking care of any background maintenance required.

```
static java.lang.Object lookup(java.lang.String name);
// Lookup method to obtain a reference to a Service within the
// local subnet and surrounding areas (last).

static java.lang.Object lookup(java.lang.String host,
                               java.lang.String name);
// Lookup method to obtain a reference to a Service on a specified host.

static java.lang.Object lookup(java.lang.String host,
                               java.lang.String name,
                               java.lang.String alias);
// Lookup method to obtain a reference to a Service on a specified
// host with a specified localised alias.
```

```
static java.lang.Object lookup(java.lang.String host,
                               java.lang.String name,
                               java.lang.String alias,
                               Tuple[] attribute);
// Lookup method to obtain a reference to a Service on a specified
// host with a specified localised alias.

static java.lang.Object lookup(java.lang.String host,
                               java.lang.String name,
                               Tuple[] attribute);
// Lookup method to obtain a reference to a Service on a specified
// host with specified attributes.

static void move(java.lang.Object service,
                 java.lang.Object target);
// Explicit move request to colocate the object referenced by service
// with the object referenced by target.

static void move(java.lang.Object service,
                 java.lang.String hostName);
// Explicit move request to relocate the object referenced by
// service to the host specified by host.

static java.lang.Object[] query(java.lang.String host);
// Query method to obtain a list of all services registered at a
// specified host.

static java.lang.Object[] query(java.lang.String host,
                                java.lang.String name);
// Query method to obtain a list of all services of a certain type
// registered at a specified host.

static void register(java.lang.Object service);
// Register method to register a generic service object.

static void register(java.lang.Object service,
                     Policy policy);
// Register method to register a generic service object with
// additional policy specification.

static void register(java.lang.Object service,
                     Policy policy,
                     java.lang.String localAlias);
// Register method to register a generic service object with
// additional policy and local alias specification.

static void register(java.lang.Object service,
                     Policy policy,
                     java.lang.String localAlias,
                     java.lang.String remoteAlias);
// Register method to register a generic service object with
// additional policy, local alias and global alias specification.
```

```
static void register(java.lang.Object service,
                     Policy policy,
                     java.lang.String localAlias,
                     java.lang.String remoteAlias,
                     Tuple[] attributes);
// Register method to register a generic service object with
// additional policy, local alias, global alias and attribute specification.

static void unbind(java.lang.Object service);
// Unbind method to remove Client reference to remote Server.
```

Attributes are specified using a `Tuple`, which defines a string-based attribute name and an object attribute value. Comparisons on boolean values and mathematical values can be done internally by the attribute matching system.

The `Tuple` class provides facilities to compare tuples and to access the components. Tuples can be serialized using the externalisable interface which is specialised for tuple values that are `java.lang.String` objects. The `Tuple` class provides the public methods shown below.

```
Tuple();
// Constructor to create an empty Tuple.

Tuple(java.lang.Object fP, java.lang.Object sP);
// Main constructor for the Tuple class.

boolean amMatch(java.lang.Object obj);
// A simple search function provided to help indexing Tuples by
// their firstPart.

boolean equals(java.lang.Object obj);
// If the object passed as a parameter is an instance of Tuple,
// returns true if the first parts are equal (tested by calling their
// equals(Object) method) and the second parts are equal.

java.lang.Object getFirstPart();
// Returns the first part of the Tuple.

java.lang.Object getSecondPart();
// Returns the second part of the Tuple.
```

B.2 Policy Interface

Policy objects define how a server object is to be treated in terms of its mobile, clone and replicate abilities. A policy is defined by the initialisation of attributes which define the required treatment.

Policy objects have publicly accessible attributes which can be set or accessed through public methods. The `Policy` class has been designed as a base policy class which, if subclassed, can have its functionality extended to define greater policies in more detail or breadth.

The API for policy specification is shown below.

```
int CLIENT;
// Defines a maximum number of clients.

int CLONE;
// Defines the object's clonable status.

int WHERE;
// Defines where objects can be sent.

int MOBILE;
// Defines whether the object is mobile.

int PROTOCOL_CHOICE;
// Defines a protocol dependency.

int REPLICATE;
// Defines the object's replicatable status.

Policy();
// Constructor for creating blank Policy objects.

void setClone(int clones);
// Sets the value of CLONE to the parameter if > 0.

boolean getClone();
// Tests whether an object can be cloned.

void setReplicate(int replicas);
// Sets the value of REPLICATE to the parameter if > 0.

boolean getReplicate();
// Tests whether an object can be replicated.

void setLocalAlias(java.lang.String[] aliasList);
// Used to set the local alias list for this service.

java.lang.String[] getLocalAlias();
// Returns the alias list allowable on the local registry.

void setRemoteAlias(java.lang.String[] aliasList);
// Used to set the remote or global alias list for this service.

java.lang.String[] getRemoteAlias();
// Returns the alias list allowable for global access.

void setProtocol(java.lang.String[] protocolList);
// Used to set the protocol list allowable for the object.
```

```
java.lang.String[] getProtocol();
// Used to get the protocol list allowable for the object.

Tuple[] getAttributes();
// Returns the attribute list currently applied to the service.

void setAttributes(Tuple[] attributeList);
// Used to set the attribute list for this service.
```

The `Registry` interface communcates the desired request to the local or specified registry through the use of protocol objects (*c.f.* Appendix A). These protocol objects operate on a registry database which contains information about all registered services within that registry's domain. A base level registry's database will contain information about all services (and aliases) directly registered with it; a primary registry will contain information about all services directly registered with it and all services registered with base level registries within its registry domain.

Registry database segments are maintained by base level registres and cooperate through a primary registry. A registry database processes request objects or different types and initiates support for additional replication and cloning support. The primary registry database is also responsible for initiating transfer of objects through the mobility support classes.

All registry databases are of the same form and extend a common interface to allow adaption between different registry levels with ease. The interface for the registry database class is shown below.

```
void bind(RegisterRequest service);
// Adds a service registration to the Registry Database.

void unbind(DeregisterRequest service);
// Removes a service registration from a Registry Database.

java.lang.Object[] listAll();
// Returns the list of all bindings within the database segment.

java.lang.Object locate(ProblemReport service);
// Returns the object matching the name specified in the ProblemReport.

java.lang.Object locate(QueryRequest service);
// Returns the object matching the QueryRequest.

java.lang.Object locateAll(QueryAllRequest service);
// Returns all objects matching the QueryAllRequest.

void suspend(ProcessRequestObject service);
// Marks an object in the database as suspended.

void move(MobileInstigationRequest service);
// Marks an object in the database as moving.
```

Different level registry databases all extend this common interface providing their own, level appropriate, constructors. The constructor for a base level registry is shown below; this constructor accepts a reference to the ORB node to which it belongs and the current location of the primary registry for its registry domain.

```
BaseRegistryData(java.net.InetAddress primary, RegistryD orbDaemon);
// Constructor for RegistryData element.
```

A primary registry maintains two registry databases: one for directly registered services and one for forwarded registration information. The constructors for the first database are shown below. The first of these constructors is used to initialise a database with a reference (local) to the primary database; the second is used to update an adapted primary registry with its currently known information set.

```
BaseRegistryData(RegistryData primary);
// Constructor for RegistryData element.

BaseRegistryData(RegistryData primary, RegistryData oldData);
// Constructor to update the status of a BaseRegistryData.
```

The constructor for the primary database is shown below. This constructor creates a new primary registry with an empty database.

```
PrimaryRegistryData()
// Constructor for RegistryData element.
```

B.3 Alias Sets

Aliases are stored within the DISCWorld system as `Alias` objects which can be used as indexes into an internal hashtable. Each alias in an alias set (belonging to the same object) maps to the one hashtable entry which contains all matching objects. This implements multiplicity and aliased names.

Aliases can be either localised aliases, in which case they are restricted to the local registry, or they can be global or remote aliases, in which case they can be propagated by the primary registry to other registry systems and may be applied to other objects. Localised aliases are not given to the primary registry so that knowledge of them does not extend past the local base registry on which they are registered.

Each alias contains a key which defines it as part of an alias set or class. This alias set is defined as the set of known names that correspond to a particular type of service. The key of an alias can be used to access a service type in a registry by using the key as the index to the hashtable which contains all service entries.

The registries are responsible for matching and combining the alias sets in a responsible manner and for managing set differences between local aliases and remote aliases. A local alias is only recognised by its local base registry, and is ignored by primary and administrative registries.

Only a local base registry will recognise a local alias otherwise it is ignored by primary registries.

Aliases have the interface shown below. The alias class implements the `java.io.Externalizable` interface which allows them to be transported in an efficient manner. It also implements the `equals(Object)` method which allows `Alias` objects to be compared relative to their contents rather than by direct instance comparisons.

```
Key key;
// The key for this Alias.

boolean exportable;
// Is true if the alias is a gobal alias.

Alias();
// Constructor to create a blank Alias.

boolean getExportable();
 // Returns true if the alias is exportable.

void setKey(Key k);
// Sets the key value for this alias to the parameter.

Key getKey();
// Returns the key value for this alias.

boolean equals(java.lang.Object obj);
// Comparison function for aliases.

void readExternal(java.io.ObjectInput in);
// Implements externalisation (reading).

void writeExternal(java.io.ObjectOutput out);
// Implements externalisation (writing).
```

A `Key` is used to define a group of objects as sharing a common attribute. In the case of Aliases, keys are used to group alias sets through a common index. Keys are used within the registry system to define alias sets. Alias sets are defined by server objects and can be local alias sets or global alias sets. Each alias set (and correspondingly each alias within the set) contains a reference to the key that matches the set. This key can then be used to access the queue of remote objects that match the alias set.

When alias sets are combined or matched, the keys for these sets have to be combined. This is done by selecting the largest set and using that key as the new key for the combined sets.

Keys for alias sets only have to be unique within a local registry. Outside of the local base registry, comparison of alias sets is performed on the value of the alias sets themselves. The `Key` class is defined below.

```
int key;
// The initialiser for all key values.

static int monitor;
// synchronisation monitor.

int hashValue;
// The hashcode of the Key.

Key();
// Creates a new Key with a new Key value.

Key(int v);
// Creates a new Key based on the provided existing value.

boolean equals(java.lang.Object obj);
// Returns true if the hash code for two keys is the same.

int getValue();
// Returns the value of the Key.

int hashCode();
// Returns the hashcode of the object
```

B.4 Communications Constructs

The DISCWorld ORB system provides specialised APIs to assist in the creation of mobile communicators and to program their semi-autonomous itineraries. Mobile communicators extend the `Mobile` interface, which provides information required for mobility and enables serialisation for transmission. The mobile interface provides mechanisms to check for program completion, result extraction and allows remote program initialisation.

Mobile objects of this form, within DISCWorld, that do not extend the provided `Mobile` interface have to ensure that they provide facilities for serialisation and correct transmission themselves. The `Mobile` interface is shown below.

```
java.lang.Boolean Completed();
// Returns true if the communicator has completed its itinerary;

ResultData returnResult()
// Returns the results from its programmed invocations.

void start();
// Initialises the communicator to restart its itinerary.
```

The `MobileCommunicator` interface, which extends `Mobile`, provides mechanisms for programming an itinerary or series of itineraries for the mobile object. The mobile object can be instructed to relocate to hosts depending on the services that are present within the registry system and to invoke methods locally once present at the remote site. A series of relocations can be combined and invocation results passed from one invocation to another through a parameter/result matching facility.

The mobile communicator interface provides methods for remote communication with clients, including methods to alter its programmed itinerary. The mobile communicator can be traced by maintaining an updated reference (a connected reference) to it; the mobile communicator can also be located through the registry system.

The API for the mobile communicator is shown below.

```
MobileCommunicator();
// Creates a blank communicator with no itinerary.

void addMethod(java.lang.String sN);
// Adds a method invocation to the itinerary with no supplied parameters.

void addMethod(java.lang.String mN,
               java.lang.Object[] p);
// Adds a method invocation to the itinerary with unnamed parameters.

void addMethod(java.lang.String mN,
               MobileParameter[] p);
// Adds a method invocation to the itinerary with parameters that may be
// defined by previous invocations.

void addMethod(java.lang.String mN,
               MobileParameter[] p,
               MobileParameter r);
// Adds a method invocation to the itinerary with parameters that may be
// defined by previous invocations, and a named result that can be used in
// future invocations.

void addItinerary(java.lang.String serverName);
// Adds an itinerary item in the form of a server to visit.

void addItinerary(Tuple[] attributes);
// Adds an itinerary item in the form of an attribute list that
// describes a server;

java.lang.Boolean Completed();
// Returns true if the communicator has completed its itinerary.

ResultData returnResult();
// Returns the results from its programmed invocations.

void run();
// Runs the current component of the communicator's itinerary.
```

```
void send();
// Sends the communicator to the next component in its itinerary.

void start();
// Initialises the communicator to restart its itinerary.
```

Parameters and return results can be matched and passed through multiple invocations through the use of specialised `MobileParameter` objects. These objects allow values to be stored (matched with a local parameter name) which will then be reused automatically by the mobile communicator. The API for a `MobileParameter` is shown below.

```
MobileParameter(java.lang.String n);
// Creates a mobile parameter with a defined local name.

MobileParameter(java.lang.String n, java.lang.Object v);
// Creates a mobile parameter with a defined local name and
// an initial value.

java.lang.String getName();
// Returns the name of the parameter.

java.lang.Object getValue();
// Returns the current value of the parameter.

void setValue(java.lang.Object v);
// Sets the current value of the parameter.
```

Appendix C

Policy Specification

Policy objects are used to specify a particular policy for a server object upon server registration. A policy object accompanies a `RegisterRequest` (*c.f.* Appendix A) and defines access to any additional functionality required by the service. Multiple policies can be created for the same server object, allowing server objects to request different policies at different stages in their lifetime.

The policy attributes defined in the base policy class are `CLIENT`, `CLONE`, `WHERE`, `MOBILE`, `PROTOCOL_CHOICE` and `REPLICATE`. Each of these attributes has default values and behaviour defined as follows.

CLIENT `CLIENT` defines the maximum number of clients that a remote object is able to concurrently accept connections from. This defines the number of connected references, not the total number of references passed out by the registry as these do not in themselves guarantee a connection. A value of -1 (default) indicates infinite clients and therefore no monitoring by the registry service; a value greater than -1 indicates the restricted number of clients. A value of 0 indicates that the server object does not wish to accept any connections.

CLONE `CLONE` defines the cloneable status of an object referencing this policy object. A value of 0 indicates not cloneable, 1 indicates cloneable for a single copy, and a value greater than 1 indicates clonable up to the number specified. A clone acts as an independent server object capable of receiving client requests.

WHERE `WHERE` defines where the clones may be sent. A value of 0 indicates that the clones are not to be distributed and are to exist on the same node as the original object. A value of 1 indicates that the clones may be distributed within the domain, and a value of 2 indicates that the clones are to be distributed throughout the object system. Global distribution will start by distributing the clones throughout the domain.

MOBILE `MOBILE` defines the mobility status of an object referencing this policy object. A value of 0 indicates that the object is stationary and a value of 1 indicates that the object is mobile. A mobile object can be sent to other nodes or relocated to other data sources or clients. When combined with the `CLONE` attribute this ability allows multiple clones to be distributed throughout the object system.

PROTOCOL_CHOICE `PROTOCOL_CHOICE` defines the protocol dependence of an object. A value of 0 indicates no protocol dependence, *i.e.* any protocol can be used to contact the object; and a value of 1 indicates protocol dependence, *i.e.* the protocols specified by `setProtocol()` are the only protocols to be used with this object.

REPLICATE `REPLICATE` defines the replicateable status of an object referencing this policy object. A value of 0 indicates that the object is not to be replicated, A value greater than 0 indicates that it is to be replicated with the number of replicas to be created equal to the current value of `REPLICATE`. All replicated objects within a replica group receive the same client requests that are sent to the original object.

Bibliography

[1] Anurag Acharya, M. Ranganathan and Joel Saltz. Sumatra: A Language for Resource-Aware Mobile Programs. In *Proceedings of the Second International Workshop on Mobile Object Systems: Towards the Programmable Internet*, pages 111–130, July 1996.

[2] Guy T. Almes, Andrew P. Black, Edward D. Lazowska and Jerre D. Noe. The Eden system: A technical review. *IEEE Transactions on Software Engineering*, Volume SE-11, Number 1, pages 43–59, January 1985.

[3] Y. Artsy, H-Y. Chang and R. Finkel. Interprocess communication in Charlotte. *IEEE Software*, Volume 4, Number 1, pages 22–28, January 1987.

[4] Y. Artsy and R. Finkel. Designing a Process Migration Facility: The Charlotte Experience. *Computer*, Volume 22, Number 9, pages 47–56, September 1989.

[5] Baruch Awerbuch, Bonnie Berger, Lenore Cowen and David Peleg. Fast Distributed Network Decompositions and Covers. *Journal of Parallel and Distributed Computing*, Volume 39, Number 2, pages 105–114, December 1996.

[6] Baruch Awerbuch and David Peleg. Online Tracking of Mobile Users. *Journal of the ACM*, Volume 42, Number 5, pages 1021–1058, September 1995.

[7] Aline Baggio. *Adaptable and Mobile-Aware Distributed Objects.* Ph.D. thesis, Université Pierre et Marie Curie, Paris, June 1999.

[8] Arno Bakker, Maarten van Steen and Andrew S. Tanenbaum. Replicated Invocations in Wide-Area Systems: A Possible Solution. In *Proceedings of the 4th Annual ASCI Conference*, pages 225–230, June 1998.

[9] Gerco Ballintijn and Maarten van Steen. Scalable Naming in Global Middleware. Technical Report IR-464, Department of Mathematics and Computer Science, Vrije Universiteit, October 1999. To be presented at PDCS'2000, Las Vegas, August 2000.

[10] Gerco Ballintijn, Maarten van Steen and Andrew S. Tanenbaum. Exploiting Location Awareness for Scalable Location-Independent. Technical Report IR-459, Department of Mathematics and Computer Science, Vrije Universiteit, January 1999.

[11] A. Barak and A. Shiloh. A Distributed Load-balancing Policy for a Multicomputer. *Software Practice & Experience*, Volume 15, Number 9, pages 901–913, September 1985.

[12] A. Barak and R. Wheeler. MOSIX: An Integrated Multiprocessor UNIX. In *Proceedings of the USENIX Winter 1989 Technical Conference*, pages 101–112, February 1989.

[13] Amnon Barak and Richard Wheeler. MOSIX for Scalable Computing Clusters. In *Mobility: Processes, Computers and Agents*, pages 54–55. Addison-Wesley, 1999.

[14] J. Baumann, F. Hohl, K. Rothermel and M. Straßer. Mole: Concepts of a Mobile Agent System. Technical Report TR-1997-15, University of Stuttgart, 1997.

[15] Bryan C. Bayderdorffer. *Associative Broadcast and the Communication Semantics of Naming in Concurrent Ssytems.* Ph.D. thesis, The University of Texas at Austin, December 1993.

[16] Bryan Bayerdorffer. An Analytical Taxonomy of Naming Systems. Technical Report TR-92-48, Department of Computer Science, University of Texas at Austin, December 1992.

[17] A. Beitz, P. King and K. Raymond. Comparing two Distributed Environments: DCE and ANSAware. In *Proceedings of International DCE Workshop*, pages 21–38, October 1993.

[18] R. Ben-Natan. *CORBA - A Guide to Common Object Request Broker Architecture.* McGraw-Hill, 1995.

[19] T. Berbers-Lee, L. Masinter and M. McCahill. Uniform Resource Locators (URL). Request For Comment RFC 1738, December 1994.

[20] Andrew D. Birrell and Bruce Jay Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, Volume 2, Number 1, pages 39–59, February 1984.

[21] Andrew Black, Norman Hutchinson, Eric Jul and Henry Levy. Object structure in the Emerald system. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 78–86, October 1986.

[22] J. Bloomer. *Power Programming with RPC.* O'Reilly & Associates, Inc., 1991.

[23] David R. Boggs. *Internet broadcasting.* Ph.D. thesis, Xerox Palo Alto Research Center, October 1983.

[24] P. Bogle and B. Liskov. Reducing Cross Domain Call Overhead Using Batched Futures. *ACM SIGPLAN Notices*, Volume 29, Number 10, pages 341–354, October 1994. Proceedings of OOPSLA'94.

[25] M. Bowman, L.L. Peterson and A. Yeatts. Univers: An Attribute-based Name Server. *Software Practice & Experience*, Volume 20, Number 4, pages 403–424, April 1990.

[26] Mic Bowman, Saumya K. Debray and Larry L. Peterson. Reasoning About Naming Systems. *ACM Transactions on Programming Languages and Systems*, Volume 15, Number 6, pages 795–825, November 1993.

[27] N. Brown and C. Kindel. *Distributed Component Object Model Protocol.* Microsoft Corporation, January 1998.

[28] Ian Buckner. UKC ANSAware Survival Guide. Internal report 5-97, University of Kent at Canterbury, January 1997. http://www.cs.ukc.ac.uk/pubs/1997/153.

[29] Luca Cardelli. Obliq: A Language with Distributed Scope. Technical Report SRC Research Report 122, Digital Equipment Corporation Systems Research Center, June 1994.

[30] Luca Cardelli. A language with distributed scope. *Computer Systems*, Volume 8, Number 1, pages 27–59, January 1995.

[31] Henri Casanova and Jack Dongarra. NetSolve: A Network Server for Solving Computational Science Problems. In *Proceedings of Supercomputing'96*, November 1996.

[32] Henri Casanova and Jack Dongarra. Using agent-based software for scientific computing in the NetSolve system. *Parallel Computing*, Volume 24, Number 12, pages 1777–1790, 1998.

[33] K. Mani Chandy, Joseph Kiniry, Adam Rifkin and Daniel Zimmerman. Webs of Archived Distributed Computations for Asynchronous Collaboration. *The Journal of Supercomputing*, Volume 11, Number 2, pages 101–118, October 1997.

[34] K. Mani Chandy, Adam Rifkin and Paolo A.G. Sivilotti. A World-Wide Distributed System Using Java and the Internet. Technical Report Caltech CS Technical Report CS-TR-96-08, Department of Computer Science, California Institute of Technology, Pasadena, CA, August 1996.

[35] David R. Cheriton and Timothy P. Mann. Decentralizing a Global Naming Service for Improved Performance and Fault Tolerance. *ACM Transactions on Computer Systems*, Volume 7, Number 2, pages 147–183, May 1989.

[36] A. Church. The calculi of lambda-conversion. *Annals of Mathematical Studies*, Volume 6, 1941. Princeton U. Press, Princeton, N.J.

[37] P.D. Coddington, K.A. Hawick, K.E. Kerry, J.A. Mathew, A.J. Silis, D.L. Webb, P.J. Whitbread, C.G. Irving, M.W. Grigg, R. Jana and K. Tang. A GIAS-compliant Server for Geospatial Image Archives. Technical Report DHPC-053, DHPC Project, University of Adelaide, 1998.

[38] P.D. Coddington, K.A. Hawick, K.E. Kerry, J.A. Mathew, A.J. Silis, D.L. Webb, P.J. Whitbread, C.G. Irving, M.W. Grigg, R. Jana and K. Tang. Implementation of a Geospatial Imagery Digital Library using Java and CORBA. In *Proceedings of Technologies of Object-Oriented Languages and Systems (TOOLS) Asia'98*, pages 280–289, September 1998.

[39] Richard Connor, Keith Sibson and Paolo Manghi. On the Unification of Persistent Programming and the World-Wide Web. In *Proceedings of the WOrkshop on the Web and Databases, part of EDBT'98, Lecture Notes in Computer Science*, Volume 1590, pages 34–51, 1998.

[40] Thinking Machines Corporation. *CM5: CM Fortran Language Reference Manual*, 1994.

[41] Giapaolo Cugola, Carlo Ghezzi, Gian Pietro Picco and Giovanni Vigna. Analyzing Mobile Code Languages. In *Proceedings of the 2nd International Workshop Mobile Object Systems - Towards the Programmable Internet, Lecture Notes in Computer Science*, Volume 1222, pages 93–110, July 1996.

[42] Karl Czajkowski, Ian Foster, Nicholas Karonis, Carl Kesselman, Stuart Martin, Warren Smith and Steven Tuecke. A Resource Management Architecture for Metacomputing Systems. *Lecture Notes in Computer Science*, Volume 1459, pages 62–82, 1998.

[43] Michael J. Demmer and Maurice P. Herlihy. The Arrow Distributed Directory Protocol. In *Distributed Computing, 12th International Symposium, DISC'98, Lecture Notes in Computer Science*, Volume 1499, pages 119–133, September 1998.

[44] J.B. Dennis and E.C. Van Horn. Programming semantics for multiprogrammed computations. *Communications of the ACM*, Volume 9, Number 3, pages 143–155, March 1966.

[45] Henry Detmold and Michael J. Oudshoorn. Communication Constructs for High Performance Distributed Computing. In *Proceedings of the 19th Australasian Computer Science Conference*, pages 252–261, January 31-February 2 1996.

[46] Peter Dickman and Mesaac Makpangou. A Refinement of the Fragmented Object Model. In *Proceedings of the Third International Workshop on Object-Orientation in Operating Systems (I-WOOOS'92)*, pages 230–234, September 1992.

[47] Peter Dickman, Mesaac Makpangou and Marc Shapiro. Contrasting Fragmented Objects with Uniform Transparent Object References for Distributed Programming. In *Proceedings of the 5th European SIGOPS Workshop, on "Models and Paradigms for Distributed Systems Structuring"*, September 1992.

[48] Peter Dömel. Integration of Java and Telescript Agents. In *Mobile Object Systems: Towards the Programmable Internet*, pages 295–314, July 1996.

[49] F. Douglis and J. Ousterhout. Transparent Process Migration: Design Alternatives and the Sprite Implementation. *Software Practice & Experience*, Volume 21, Number 8, pages 757–785, August 1991.

[50] R.S. Fabry. Capability-Based Addressing. *Communications of the ACM*, Volume 17, Number 7, pages 403–412, July 1974.

[51] Katrina E.K. Falkner. A Model of Location Transparency. In *Proceedings of the 7th Integrated Data Environments Australia Workshop*, February 2000.

[52] Katrina E.K. Falkner, Paul D. Coddington and Michael J. Oudshoorn. Implementing Asynchronous Remote Method Invocation in Java. In *Proceedings of PART'99*, pages 22–34, 1999.

[53] Alan Fekete, M. Frans Kaashoek and Nancy Lynch. Implementing Sequentially Consistent Shared Objects using Broadcast and Point-To-Point Communication. *Journal of the ACM*, Volume 45, Number 1, pages 35–69, January 1998.

[54] Steven Fitzgerald, Ian Foster, Carl Kesselman, Gregor von Laszewski, Warren Smith and Steven Tuecke. A Directory Service for Configuring High-Performance Distributed Computations. In *Proceedings of the 6th IEEE Symposium on High-Performance Distributed Computing*, 1997.

[55] D. Flanagan. *Java in a Nutshell.* O'Reilly & Associates, Inc., 1996.

[56] Paola Flocchini, Bernard Mans and Nicola Santoro. Sense of Direction in Distributed Computing. In *Distributed Computing, 12th International Symposium, DISC '98*, pages 1–15, September 1998. Lecture Notes in Computer Science 1499.

[57] Ian Foster and Carl Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *International Journal of Supercomputer Applications and High Performance Computing*, Volume 11, Number 2, pages 115–128, 1997.

[58] Ian Foster and Carl Kesselman. The Globus Project: A Status Report. In *Proceedings of Heterogeneous Computing Workshop*, pages 4–18, 1998.

[59] Ian Foster, Carl Kesselman and Steven Tuecke. The Nexus Approach to Integrating Multithreading and Communication. *Journal of Parallel and Distributed Computing*, Volume 37, Number 1, pages 70–82, August 1996.

[60] Ian Foster, David R. Kohr, Robert Olsen, Steven Tuecke and Ming Q. Xu. *Languages, Compilers, and Run-time Systems for Scalable Computers*, Chapter Point-to-Point Communication Using Migrating Ports, pages 199–212. Kluwer Academic Publishers, 1995.

[61] Robert J. Fowler. *Decentralized Object Finding Using Forwarding Addresses*. Ph.D. thesis, University of Washington, December 1985. Department of Computer Science technical report 85-12-1.

[62] Stan Franklin and Art Graesser. Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents. In *Proceedings of the Third International Workshop on Agent Theories, Architectures, and Languages*, pages 21–36, 1996.

[63] A.G. Fraser. On the Meaning of Names in Programming Systems. *Communications of the ACM*, Volume 14, Number 6, pages 409–416, June 1971.

[64] N. Gehani and W. Roome. Concurrent C. *Software Practice & Experience*, Volume 16, Number 9, pages 821–844, September 1986.

[65] D. Gelernter. Generative communication in linda. *ACM Transactions on Programming Languages and Systems*, Volume 7, Number 1, pages 80–112, January 1985.

[66] A. Gokhale and D. Schmidt. Evaluating CORBA Latency and Scalability Over High-Speed ATM Networks. In *Proceedings of the 17th International Conference on Distributed Computing Systems*, 1997.

[67] Aniruddha Gokhale and Douglas C. Schmidt. Principles for Optimising CORBA Internet Inter-ORB Protocol. In *Proceedings of HICSS Conference*, January 1998.

[68] A. Goscinkski. *Distributed Object Systems: The Logical Design.* Addison-Wesley Publishing Company, 1991.

[69] J. Gosling and H. McGilton. The Java Language Environment: A White Paper. Technical report, Sun Microsystems, October 1995.

[70] James Gosling, Bill Joy and Guy Steele. *The Java Language Specification*. JavaSoft Series. Addison Wesley Longman, 1996.

[71] Robert S. Gray. Agent Tcl: A Transportable Agent System. In *Proceedings of the CIKM Workshop on Intelligent Information Agents, Fourth International Conference on Information and Knowledge Management (CIKM '95)*, 1995.

[72] Robert S. Gray. Agent Tcl: A flexible and secure mobile-agent system. In *Proceedings of the Fourth Annual Usenix Tcl/Tk Workshop*, pages 9–23, 1996.

[73] Andrew S. Grimshaw, Michael J. Lewis, Adam J. Ferrari and John F. Karpovich. Architectural Support for Extensibility and Autonomy in Wide-Area Distributed Object Systems. Technical Report CS-98-12, Department of Computer Science, University of Virginia, June 1998.

[74] Andrew S. Grimshaw and William A. Wulf and the Legion team. The Legion Vision of a Worldwide Virtual Computer. *Communications of the ACM*, Volume 40, Number 1, pages 39–45, January 1997.

[75] Andrew S. Grimshaw, William A. Wulf, James C. French, Weaver Alfred C and Paul F. Reynolds. Legion: The Next Logical Step Toward a Nationwide Virtual Computer. Technical Report CS-94-21, Department of Computer Science, University of Virginia, June 1994.

[76] A.S. Grimshaw, E.C. Loyot and J.B. Weissman. *Mentat Programming Language (MPL) Reference Manual*. Department of Computer Science, University of Virginia, cs-91-32 edition, November 1991.

[77] James D. Guyton and Michael F. Schwartz. Locating Nearby Copies of Replicated Internet Servers. In *Proceedings of SIGCOMM'95*, pages 288–298, 1995.

[78] K.A. Hawick, P.D. Coddington, D.A. Grove, J.F. Hercus, H.A. James, K.E. Kerry, J.A. Mathew, C.J. Patten, Andrew Silis and F.A. Vaughan. DISCWorld: An Environment for Service-Based Metacomputing. *Future Generations of Computer Science*, Volume 15, pages 623–635, 1998.

[79] K.A. Hawick and H.A. James. A Distributed Job Placement Language. Technical Report DHPC-070, Distributed High Performance Computing Group, Department of Computer Science, University of Adelaide, May 1999. Available from http://www.dhpc.adelaide.edu.au/reports/070/abs-070.html.

[80] K.A. Hawick and H.A. James. Protocols, Tools and Issues for Distributed Computing Network Monitoring and Reconfiguration. Technical Report DHPC-068, Distributed High Performance Computing Group, Department of Computer Science, University of Adelaide, May 1999. Available from http://www.dhpc.adelaide.edu.au/reports/068/abs-068.html.

[81] K.A. Hawick, H.A. James and J.A. Mathew. Remote Data Access in Distributed Object-Oriented Middleware. *Journal of Parallel and Distributed Computing Practices, Special Issue on Distributed Object Oriented Systems*, 2000. To appear.

[82] K.A. Hawick, H.A. James, C.J. Patten and F.A. Vaughan. DISCWorld: A Distributed High Performance Computing Environment. In *Proceedings of High Performance Computing and Networking (HPCN) Europe '98, Amsterdam*, pages 598–606, April 1998.

[83] Michi Henning. Binding, Migration, and Scalability in CORBA. *Communications of the ACM*, Volume 41, Number 10, pages 62–71, October 1998.

[84] Maurice Herlihy. The Aleph Toolkit: Support for Scalable Distributed Shared Objects. In *Proceedings of CANPC'99*, pages 137–149, 1999.

[85] Maurice Herlihy and Michale P. Warres. A Tale of Two Directories: Implementing Shared Objects in Java. In *Proceedings of ACM Java Grande'99*, 1999.

[86] M.P. Herlihy. The Aleph Toolkit: Platform-independent distributed shared memory (preliminary report). http://www.cs.brown.edu/˜mph/aleph, 1999.

[87] S. Hirano. HORB: Distributed Execution of Java Programs. In *Worldwide Computing And Its Applications, Lecture Notes in Computer Science*, Volume 1274, pages 29–42, 1997.

[88] C. Hoare. Communicating sequential processes. *Communications of the ACM*, Volume 21, Number 8, pages 666–677, August 1978.

[89] Anatol W. Holt. Program Organization and Record Keeping for Dynamic Storage Allocation. *Communications of the ACM*, Volume 4, Number 10, pages 422–431, October 1961.

[90] B.C. Housel and D.B. Lindquist. WebExpress: A System for Optimizing Web Browsing in a Wireless Environment. In *Proceedings of the Second Annual International Conference on Mobile Computing and Networking*, pages 108–116, November 1996.

[91] Norman C. Hutchinson. *Emerald: An Object-Based Language for Distributed Programming.* Ph.D. thesis, University of Washington, January 1987. Department of Computer Science technical report 87-01-01.

[92] J.K. Iliffe and Jane G. Jodeit. A dynamic storage allocation scheme. *The Computer Journal*, Volume 5, pages 200–209, October 1962.

[93] J. Ioannidis and G.Q. Maguire. The Design and Implementation of a Mobile Internetworking Architecture. In *Proceedings of the USENIX Winter 1993 Technical Conference*, pages 491–502, January 1993.

[94] Christian Ionitoiu. Designing agents for archie and ftp sessions in Obliq. In *Proceedings of the 2nd ECOOP Workshop on Mobile Object Systems*, pages 49–51, July 1996.

[95] Matthew Izatt, Patrick Chan and Tim Brecht. Ajents: Towards an Environment for Parallel, Distributed and Mobile Java Appilcations. In *Proceedings of the ACM Symposium on Principles and Practice of Parallel Programming*, 1999.

[96] J.A.Mathew, P.D.Coddington and K.A.Hawick. Analysis and Development of Java Grande Benchmarks. In *Proceedings of the ACM 1999 Java Grande Conference*, June 1999.

[97] H.A. James, K.A. Hawick and P.D. Coddington. Scheduling Independent Tasks on Metacomputing Systems. In *Proceedings of Parallel and Distributed Computing Systems (PDCS'99)*, pages 156–162, August 1999.

[98] Heath A. James. *Scheduling in Metacomputing Systems.* Ph.D. thesis, Department of Computer Science, University of Adelaide, July 1999.

[99] Jin Jing, Abdelsalam (Sumi) Helal and Ahmed Elmagarmid. Client-Server Computing in Mobile Environments. *ACM Computing Surveys*, Volume 31, Number 2, pages 117–157, June 1999.

[100] A.D. Joseph, J.A. Tauber and M.F. Kaashoek. Mobile Computing with the Rover Toolkit. *IEEE Transactions on Computers*, Volume 46, Number 3, pages 337–352, March 1997.

[101] ISO/IEC JTC1/SC21. *N8125: Draft Recommendation X.903: Basic Reference Model of Open Distributed Processing - Part 3: Prescriptive Model*, June 1993.

[102] Eric Jul, Henry Levy, Norman Hutchinson and Andrew Black. Fine-Grained Mobility in the Emerald System. *ACM Transactions on Computer Systems*, Volume 4, Number 1, pages 109–133, February 1988.

[103] K. E. Kerry and K. A. Hawick. Kriging Interpolation on High-Performance Computers. In *Proceedings of High-Performance Computing and Networks (HPCN) 98*, pages 429–438, April 1998.

[104] K. E. Kerry and K. A. Hawick. Service Management in DISCWorld using CORBA. In *Proceedings of the Fifth IDEA Workshop*, pages 18–23, February 1998.

[105] Graham Kirby, Ron Morrison and David Stemple. Linguistic Reflection in Java: A Quantative Assessment. *Software Practice & Experience*, Volume 28, Number 10, pages 1045–1077, 1998.

[106] Graham N. C. Kirby. *Reflection and Hyper-Programming in Persistent Programming Systems.* Ph.D. thesis, University of St Andrews, 1992.

[107] J.J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. *ACM Transactions on Computer Systems*, Volume 10, Number 1, pages 3–25, February 1991. Previous version appeared at the 13th Symposium on Operating System Principles in Pacific Grove, CA in October 1991.

[108] Michael M. Kong. DCE: An Environment for Secure Client/Server Computing. *Hewlett-Packard Journal*, pages 6–15, December 1995.

[109] Micheal M. Kong and David Truong. DCE Directory Services. *Hewlett-Packard Journal*, pages 23–27, December 1995.

[110] D. Kotz, R. Gray, S. Nog, D. Rus, S. Chawla and G. Cybenko. AGENT TCL: Targeting the Needs of Mobile Computers. *IEEE Internet Computing*, Volume 1, Number 4, pages 58–67, July/August 1997.

[111] Bobby Krupczak, Ken Calvert and Mostafa Ammar. Implementing Protocols in Java: The Price of Portability. Technical Report GIT-CC-97-21, College of Computing, Georgia Institute of Technology, August 1997.

[112] Butler W. Lampson. Designing a Global Name Service. In *Proceedings 1986 Conference on Principles of Distributed Computing, Ontario*, pages 1–10, August 1986.

[113] D. B. Lange. Mobile Objects and Mobile Agents: The Future of Distributed Computing? *Lecture Notes in Computer Science*, Volume 1445, pages 1–12, 1998.

[114] D. B. Lange. Present and Future Trends of Mobile Agent Technology. *Lecture Notes in Computer Science*, Volume 1477, pages 1, 1998.

[115] Danny B. Lange, Mitsuru Ochima, Gunter Karjoth and Kazuya Kosaka. Aglets: Programming Mobile Agents in Java. *Lecture Notes in Computer Science*, Volume 1274, pages 253–266, 1997.

[116] Kai Li and Paul Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transaction on Computer Systems*, Volume 7, Number 4, pages 321–359, November 1989.

[117] B. Liskov and L. Shrira. Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems. In *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation*, pages 260–267, June 1988.

[118] Nancy A. Lynch and Mark R. Tuttle. An Introduction to Input/Output Automata. *CWI Quarterly*, Volume 2, Number 3, pages 219–246, September 1989.

[119] Jason Maassen, Rob van Nieuwpoort, Ronald Veldema, Henri E. Bal and Aske Plaat. An Efficient Implementation of Java's Remote Method Invocation. In *Proceedings of the ACM Symposium on Principles and Practice of Parallel Programming*, pages 173–182, May 1999.

[120] General Magic. Telescript Language Reference. General Magic, October 1995.

[121] J. Maisonneuve. *Hobbes: un modéle de liason de références réparties.* Ph.D. thesis, Université Pierre et Marie Curie, October 1996.

[122] Mesaac Makpangou, Yvon Gourhant, Jean-Pierre Le Narzul and Marc Shapiro. *Readings in Distributed Computing Systems*, Chapter Fragmented Objects for Distributed Abstractions, pages 170–186. IEEE Computer Society Press, 1991.

[123] Mesaac Makpangou, Yvon Gourhant, Jean-Pierre Le Narzul and Marc Shapiro. Structuring distributed applications as fragmented objects. Technical Report 140, INRIA, Rocquencourt, France, January 1991.

[124] Keith A. Lantz Marvin M. Theimer and David R. Cheriton. Preemptable Remote Execution Facilities for the V-System. In *Proceedings of the 10th ACM Symposium on Operating System Principles*, pages 2–12, December 1985.

[125] J.A. Mathew, A.J. Silis and K.A. Hawick. Inter Server Transport of Java Byte Code in a Metacomputing Environment. In *Proceedings of Technology of Object-Oriented Languages and Systems Pacific (TOOLS 28)*, pages 264–277, November 1998.

[126] M. Metcalf and J. Reid. *Fortran 90 Explained.* Oxford, 1990.

[127] Microsoft Corporation, Digital Equipment Corporation. *The Component Object Model Specification*, October 1995.

[128] D. Milojičić, M. Breugst, I. Busse, J. Campbell, S. Covaci, B. Friedman, K. Kosaka, D. Lange, K. Ono, M. Oshima, C. Tham, S. Virdhagriswaran and J. White. MASIF: The OMG Mobile Agent System Interoperability Facility. In K. Rothermel and F. Hohl (editors), *Proceedings of the 2nd International Workshop on Mobile Agents, Lecture Notes in Computer Science*, Volume 1477, pages 50–67, 1998.

[129] Dejan Milojičić, Fred Douglis, Yves Paindaveine, Richard Wheeler and Songnian Zhou. Process Migration. Technical Report HPL-1999-21, Computer Systems Laboratoty, HP Laboratories Palo Alto, February 1999.

[130] Dejan Milojičić, Frederick Douglis and Richard Wheeler (editors). *Mobility: Processes, Computers, and Agents*. Addison-Wesley, 1999.

[131] D.S. Milojičić, D. Chaufan and W. laForge. Mobile Objects and Agents (MOA), Design, Implementation and Lessons Learned. In *Proceedings of the 4th USENIX Conference on Object-Oriented Technologies (COOTS'98)*, pages 179–194, April 1998.

[132] Luc Moreau. A Distributed Garbage Collector with Diffusion Tree Reorganisation and Mobile Objects. *ACM Sigplan Notices*, Volume 34, Number 1, pages 204–215, January 1999.

[133] Luc Moreau, David DeRoure and Ian Foster. NeXeme: a Distributed Scheme Based on Nexus. In *Third International Europar Conference (EURO-PAR'97)*, pages 581–590, August 1997. Lecture Notes in Computer Science, volume 1300.

[134] Scott Oaks and Henry Wong. *Java Threads*. Nutshell Handbook. O'Reilly & Associates, Inc., United States of America, 1st edition, 1997.

[135] Object Management Group (OMG). Naming Service Specification. CORBAServices Specification, Available from http://www.omg.org/, March 1995.

[136] Object Management Group (OMG). Common Object Services Specification. Available from http://www.omg.org/, March 1997.

[137] Object Management Group (OMG). Trading Object Service Specification. CORBAServices Specification, Available from http://www.omg.org/, March 1997.

[138] Object Management Group (OMG). The Common Object Request Broker: Architecture and Specification (Revision 3.0). Framingham, MA, July 1998.

[139] Department of Defence. *Reference manual for the Ada programming language*. DoD, Washington, ansi/mil-std-1815a edition, January 1983.

[140] Home Page of the Open Group. Available at http://www.opengroup.org/.

[141] M.A. Oliver and R. Webster. Kriging: a Method of Interpolation for Geographical Information Systems. *International Journal Geographic Information Systems*, Volume 4, Number 3, pages 313–332, 1990.

[142] J.K. Ousterhout, A.R. Cherenson, F. Douglis, M.N. Nelson and B.B. Welch. The Sprite network operating system. *IEEE Computer*, Volume 22, Number 9, pages 47–56, September 1988.

[143] Craig Partridge, Trevor Mendez and Walter Milliken. Host anycasting service. Request For Comment RFC 1546, November 1993.

[144] C.J. Patten, K.A. Hawick, J.F. Hercus and A.L. Brown. Distributed and Hierarchical Storage Systems. In *Proceedings of the 6th IDEA Workshop, Rutherglen*, pages 57–62, January 1999.

[145] Craig J. Patten, K.A. Hawick and J.F. Hercus. Towards a Scalable Metacomputing Storage Service. In *Proceedings of High Performance Computing and Networks (HPCN) Europe'99*, pages 350–359, April 1999.

[146] C.E. Perkins. Mobile Networking with Mobile IP. *IEEE Internet Computing*, Volume 2, Number 1, pages 58–69, January/February 1998.

[147] David Plainfossé and Marc Shapiro. A Survey of Distributed Garbage Collection Techniques. In *International Workshop on Memory Management (IWMM95)*, pages 211–249, 1995. Lecture Notes in Computer Science, volume 986.

[148] Jon Postel. Internet Protocol. Request For Comment RFC 791, September 1981.

[149] M. Powell and B. Miller. Process Migration in DEMOS/MP. In *Proceedings of the 9th ACM Symposium on Operating Systems Principles*, pages 110–119, October 1983.

[150] D. Ramazani, G.V. Bochmann and P. Flocchini. Object Naming and Object Composition. Technical Report 1135, Universitè de Montrèal, November 1998.

[151] M. Ranganathan, A. Acharya, D.D. Sharma and Saltz J. Network-aware Mobile Programs. In *Proceedings of the USENIX 1997 Conference*, pages 91–103, January 1998.

[152] Karunaharan Ratnam, Ibrahim Matta and Sampath Rangarajan. A Fully Distributed Location Management Scheme for Large PCS. Technical Report BU-CS-99-010, Boston University, August 1999.

[153] Jonathan Rees and William Clinger. Revised Report on the Algorithmic Language Scheme. *Lisp Pointers*, Volume 4, Number 3, pages 1–55, September 1991.

[154] P. Roe and C. Szyperski. Transplanting in Gardens: Efficient Heterogeneous Task Migration for Fully Inverted Software Architectures. In *Proceedings of the 4th Australasian Computer Architecture Conference (ACAC'99), Auckland, New Zealand*, January 1999.

[155] William Rubin and Marshall Brain. *Understanding DCOM.* P T R Prentice-Hall, 1999.

[156] Daniela Rus, Robert Gray and David Kotz. Transportable Information Agents. In *International Conference on Autonomous Agents*, pages 228–236. ACM Press, February 1997.

[157] V. Ryan, S. Seligman and R. Lee. Schema for Representing Java(tm) Objects in an LDAP Directory. Internet-Draft, February 1999.

[158] J.H. Saltzer. *Operating Systems - An Advanced Course*, Chapter Naming and Binding of Objects, pages 99–208. Springer-Verlag, 1978. Also in Lecture Notes in Computer Science, volume 60.

[159] J.H. Saltzer. On The Naming and Binding of Network Destinations. Request For Comment RFC 1498, August 1993.

[160] D. Schmidt, M. Kircher and I. Pyarali. Location Forwarding. Available at http://www.cs.wustl.edu/~schmidt/ACE_wrappers/TAO/docs/forwarding.html, 1998.

[161] Michael D. Schroeder, Andrew D. Birrell and Roger M. Needham. Experience with Grapevine: The Growth of a Distributed System. *ACM Transactions on Computer Systems*, Volume 2, Number 1, pages 3–23, February 1984.

[162] Giovanna Di Marzo Serugendo, Murhimanya Muhugusa and Christian F. Tschudin. A Survey of Theories for Mobile Agents. *WWW Journal*, Volume 1, Number 3, pages 139–153, 1998.

[163] Marc Shapiro. A Binding Protocol for Distributed Shared Objects. In *Proceedings of the 14th International Conference on Distributed Computer Systems (ICDCS)*, pages 134–141, June 1994.

[164] Marc Shapiro, Peter Dickman and David PlainFossé. SSP Chains: Robust, Distributed References Supporting Acyclic Garbage Collection. Technical Report 1799, INRIA, Rocquencourt, France, November 1992.

[165] J.W. Stamos and D. K. Gifford. Remote Evaluation. *ACM Transactions on Programming Languages and Systems*, Volume 12, Number 4, pages 537–565, October 1990.

[166] Sun Microsystems, Inc. Java Core Reflection API and Specification. Available from http://www.java.sun.com/products/jdk/1.3/docs/guide/reflection/spec/java-reflectionTOC.doc.html, February 1998.

[167] Sun Microsystems, Inc. Java Remote Method Invocation Specification. Available from http://www.sun.com/, October 1998.

[168] Sun Microsystems, Inc. JNDI: Java Naming and Directory Interface. Available from http://www.sun.com/, January 1998.

[169] Sun Microsystems, Inc. Jini Architecture Specification. Available from http://www.sun.com/jini, January 1999.

[170] Sun Microsystems, Inc. Jini Distributed Leasing Specification. Available from http://www.sun.com/jini, January 1999.

[171] Sun Microsystems, Inc. Jini Lookup Service Specification. Available from http://www.sun.com/jini, January 1999.

[172] Hideki Tai and Kazuya Kosaka. The Aglets Project. *Communications of the ACM*, Volume 42, Number 3, pages 100–101, March 1999.

[173] D. Terry, M. Theimer, K. Petersen, A. Demers, M. Spreitzer and C. Hauser. Managing Update Conflicts in a Weakly Connected Replicated Storage System. In *Proceedings of the 15th ACM Symposium on Operating System Principles*, pages 173–183, December 1995.

[174] D.B. Terry. *Distributed Name Servers: Naming and Caching in Large Distributed Computing Environments.* Ph.D. thesis, University of California, Berkely, 1985. Available as UCB/CSD Tech. Rep 85-228 and as Xerox PARC Tech. Rep. CSL-85-1.

[175] Thinking Machines Corporation. *The Connection Machine CM5 Technical Summary*, 1991.

[176] Maarten van Steen and Franz J. Hauck. Algorithmic Design of the Globe Wide-Area Location Service. Technical Report IR-440, Vrije Universiteit, December 1997.

[177] Maarten van Steen, Franz J. Hauck, Philip Homburg and Andrew S. Tanenbaum. Locating Objects in Wide-Area Systems. *IEEE Communications*, Volume 36, Number 1, pages 104–109, January 1998.

[178] Maarten van Steen, Phillip Homburg and Andrew S. Tanenbaum. The Architectural Design of Globe: A Wide-Area Distributed System. Technical Report IR-422, Vrije Universiteit, March 1997.

[179] Maarten van Steen, Andrew S. Tanenbaum, Ihor Kus and Henk J. Sips. A Scalable Middleware Solution for Advanced Wide-Area Web Services. Technical Report IR-446, Vrije Universiteit, March 1998.

[180] Bill Venners. *Inside the Java Virtual Machine.* Computing McGraw-Hill, 1998.

[181] T. von Eicken, D.E.Culler, S.C. Goldstein and K.E. Schauser. Active messages: a mechanism for integrated communication and computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture (ISCA'92)*, pages 256–267, May 1992.

[182] Gregor von Laszewski and Ian Foster. Usage of LDAP in Globus. Available at http://www-fp.globus.org/documentation/papers.html, April 1998.

[183] E. F. Walker, R. Floyd and P. Neves. Asynchronous Remote Operation Execution In Distributed Systems. In *Proceedings of the Tenth International Conference on Distributed Computing Systems*, pages 125–136, May/June 1990.

[184] J.E. White. Mobile Agents. General Magic, 1995.

[185] David Wong, Noemi Paciorek and Dana Moore. Java-based Mobile Agents. *Communications of the ACM*, Volume 42, Number 3, pages 92–102, March 1999.

[186] W. Yeong, T. Howes and S. Kille. Lightweight Directory Access Protocol. Request For Comment RFC 1777, March 1995.