Issues in Persistent Systems

Eva Z. Bem

Persistent Systems Research Group
Basser Department of Computer Science
University of Sydney
ewa@cs.usyd.edu.au

Abstract

Four systems were investigated to determine by what means they support persistence. This investigation lead to realisation that, except for the protection mechanism, no consensus exists in the remaining areas. Each of the systems takes a different approach to provision of persistence, and there are no established criteria to decide which one offers the most suitable solution. Some of the unresolved issues are: the size and structure of the primitive persistent objects and their addressing mechanism, the means by which the persistent computations are supported, the design of the persistent kernel, the impact of the primitive object definition on stability and resilience, etc. This paper is not an attempt to offer the definitive answers, but rather to formulate the pertinent questions which may eventually lead to such answers.

1 Introduction

The term persistent system is used in this paper to describe the systems where persistence is provided at the operating system level, either by purely software means, or with varying levels of support from hardware. There is a number of requirements an operating system has to fulfil in order to support orthogonal persistence [2]:

- Persistent objects, consisting of data and relationships with other persistent objects, have to be supported as the basic abstraction of the system. Access to objects is to be based on a uniform addressing scheme.
- Persistent computations have to be supported by containing the state of processes within the persistent objects.
- A protection mechanism for persistent objects has to be provided.
- Persistent objects must be both stable and resilient, that is they must survive system malfunctions. The transition of objects between long and short term memory, and any other mechanisms used to ensure persistence must be transparent to users.

Persistent systems, which treat all system resources as objects, typically use a capability-based protection mechanism. Access control lists which require a well-defined notion of a user do not fit the persistence paradigm. This is one area of persistent system design which has an established and sufficiently justified solution, therefore it will not be discussed further.

2 Systems Selected for Case Study

The following systems were selected to provide a representative sample of hardware and system software mechanisms which can be employed in support of persistence:

- IBM AS/400 [12] - a commercial system with custom built hardware,
• KeyKOS [4] - a commercial system designed to run with no support from hardware,
• Monads/PC, Monads/MM [10] - experimental systems based on custom built hardware,
• Grasshopper [3, 8] - an experimental system based on 64-bit conventional hardware.

For the AS/400 system the design goal was to build a data processing environment of high availability. The architecture is based on the application programming interface. It constitutes a boundary above which no details of hardware implementation are visible. Persistence in AS/400 is merely a tool to achieve this goal. As a result the lower layers of the system support persistence, but the view offered at the application level is strictly conventional.

The design of the KeyKOS nanokernel was motivated by the need to provide security, reliability and round-the-clock availability. These goals were achieved by implementing orthogonal persistence in the system kernel. KeyKOS ran on a variety of IBM System/370 computers. It is no longer in production.

The other two systems, Monads and Grasshopper, are purely experimental. The aim of the Monads project was to develop techniques for supporting software engineering, and to improve data security. The architecture removes the distinction between short and long-term data providing a foundation for a persistent computing environment. The Monads project has now been abandoned, mostly because it was not possible for its hardware to match the performance levels achieved commercially.

Grasshopper is an operating system designed to run on conventional hardware to avoid the problems encountered by the Monads project. Unlike the other three systems, Grasshopper was expressly designed to provide a base for experimentation in persistence.

3 Persistent Objects

The persistent store and its objects are defined by a number of attributes ie. model of persistence (how objects become persistent), object structure (what constitutes a persistent object), and object identification and addressing (how objects are located).

There are three known models of persistence [13]: by designation, by embedding, and by reachability. The first model leaves the decision of data persistence to the user, and as such does not satisfy the requirements of orthogonal persistence. The embedded persistence relies on the concept of a single-level store, where each object has a unique address which is never reallocated. AS/400 and KeyKOS represent this model. Persistence by reachability is based on the assumption, that an object which cannot be reached, cannot be used, and can be deleted. Grasshopper and Monads represent a mixed model of persistence. Large grain objects persist because they are embedded in persistent address space. The building blocks of the objects, virtual pages, persist as long as they can be reached from the global root of persistence. It should be noted that a virtual page as such is not an object in either system.

In each of the selected systems the structure of the basic persistent object is different. At the lowest level though each one of them uses the paged virtual memory mechanism provided by hardware to support movement of objects between stable and volatile storage. Both AS/400 and KeyKOS define their objects directly in terms of their single address space. In AS/400 a primitive object is a 2^{24} byte segment, which is used as a building block for high level objects. Objects are never completely destroyed, the object header remains on disk, and its address is
never reused. In KeyKOS the primitive objects are 4 kbyte pages and nodes mapped directly to virtual pages. Pages are used to store data, and nodes to store capabilities. All higher level objects are constructed using these two primitive objects, therefore the virtual memory mechanism of the host hardware directly supports the manipulations on the primitive objects.

The definition of a persistent object in Monads and Grasshopper is only remotely related to their hardware representation. In Monads a primitive persistent object is a variable size segment. A group of segments constitutes a module. Modules reside in unique address spaces, which are never reused. Segments are orthogonal to page boundaries, they may have arbitrary sizes, up to the total size of an address space. In Grasshopper, a persistent object is either a container (the abstraction over data) or a locus (the abstraction over execution). Each object is represented by one or more pages in the kernel address space. The data in a container resides in the container’s address space. The size of data may change dynamically.

In persistent systems no assumptions can be made about the longevity of objects. This implies that the system has to provide means of addressing a very large (in theory - an infinite) number of objects. The hardware address size may be extended by software means with address translation or swizzling. The cost of address manipulation depends heavily on the characteristics of the object [9], and none of the two techniques has a clear advantage.

KeyKOS, having no support from hardware, uses a form of swizzling. In its early implementation, the AS/400 system used a 48-bit processor but 64-bit object pointers, and it had to resort to address translation. Current models of AS/400 use a 64-bit PowerPC processor, and the object identifiers (OIDs) are equivalent to virtual addresses. Monads/PC and Monads/MM had custom built processors, which provided 60 and 128-bit addressing respectively, and used a one-to-one mapping between OIDs and virtual addresses. Grasshopper was designed to run on a 64-bit processor (currently Digital Alpha), and it also uses a direct one-to-one mapping between OIDs and virtual addresses. The Alpha processor actually offers a 43-bit wide address which appears large enough for an experimental system. Some researchers believe [1, 6] that even a 64-bit address may not be sufficient in the long term.

Because of the lack of experimental evidence, it is not clear what address space size is sufficiently large for a persistent system. Extending the hardware address size with software techniques may have a detrimental effect on overall performance, so addressing issues remain a subject of continuing research in the area of persistence.

4 Persistent Computations

The support for persistent computations marks the difference between orthogonally persistent systems and systems which merely provide a persistent store. Persistent processes introduce an additional level of complexity to the system, because of their interactions with the system kernel where the distinction between volatile and stable data remains visible.

In a conventional system all kernel data, including all information related to processes, is transient, and has to be rebuilt every time the system is rebooted. It has been suggested [2] that if this data is made persistent, the computations become persistent implicitly. The problem is that the state of many of the system resources is inherently transient, and cannot survive the shutdown, for example main memory, I/O devices, network connections, etc. If the data representing the transient state of the kernel survives
shutdown, on restart it would have to be reconstructed to reflect the actual state of the
system. The remaining kernel data may depend in some ways on the data structures built
during the previous bootstrap. These structures would also have to be rebuilt to match the
new state of the system, and so on. The kernel code itself is also transient. A process
stopped while executing within the old kernel code cannot simply continue execution
within the new code after an upgrade.

Since a core of transient data, however small, always exists, it is not possible to construct a
fully persistent kernel. Therefore, when designing a persistent kernel, the fundamental
decision is where to place the division line between the kernel transient and persistent data
[11], and how to ensure their isolation. In particular, the state of a persistent process must
not be captured while interacting with kernel transient data. There are two broad
approaches to handle this situation:

- no persistent processes are allowed to execute within the kernel space; if a kernel
  service is required a request is made to a transient kernel server process, while the
  persistent process waits for the return value outside the kernel,
- persistent processes are allowed into the kernel space, but must leave it before their
  state is captured.

Of the four discussed systems, only the AS/400 does not support persistent processes. The
KeyKOS system is based on a stateless nanokernel, and it requires no dynamic allocation
of storage. Any data used by the nanokernel can be fully reconstructed from the persistent
nodes and pages. As some state information cannot be saved in persistent objects [7],
some undisclosed ad hoc mechanisms are used to resynchronise the system after restart.

The Monads kernel [5] is implemented as a set of transient processes each performing a
specific task. User processes make requests to kernel processes, and suspend until the
request is complete, so they never interact directly with kernel data structures.
Grasshopper uses two techniques to prevent persistent loci from becoming dependent on
transient kernel data. Loci are allowed into the kernel for servicing calls which do not
involve delays. All other calls, mostly the I/O requests, are handled by dedicated transient
kernel loci, as in Monads.

5 Stability and Resilience

Data is considered stable if it can survive system restarts and failures. Therefore stability of data
relates directly to the stability of physical storage media. The simple definition of objects in
AS/400 and KeyKOS ensures that stability of objects is equivalent to stability of virtual pages.
KeyKOS achieves it by storing modified pages in a log on disk. AS/400 uses uninterruptible
power source (UPS), and another technique called a continuously powered main store (CPM),
to ensure that the main memory and data residing in it can be considered stable. Both Monads
and Grasshopper also provide stability to virtual pages, using a variety of shadow paging
techniques. The mapping between virtual pages and persistent objects in both systems is
not straightforward, and may change dynamically. This introduces additional complexity
to stabilising their basic persistent objects.

A store is resilient if a consistent state of data in it is always recoverable despite system failures.
The technique of choice, used in all the discussed systems except for AS/400, is a form of
asynchronous incremental checkpoint, a software technique developed originally for data
bases. A variety of checkpointing methods has been tried in Grasshopper, including global
checkpoint enhanced with hardware techniques, and optimistic checkpoint strategies. The aim
was to minimise the performance loss associated with checkpointing. For resilience the AS/400 relies on its ability to perform an orderly system shutdown, when all modified pages are copied from memory to hard disks. This can be viewed as a form of a checkpoint. It is important to note that such a checkpoint does not affect the normal error-free operation of the system.

6 Conclusion

Many issues related to persistent systems are still unresolved, and remain in the domain of research and experimentation. The only exception is the protection mechanism, with general agreement, that persistence is best served by capability-based systems. There is no consensus on the optimal size and structure of a persistent object, the best mechanisms for addressing objects, the issues related to persistence of processes and designing persistent kernels, and the efficient way for providing stability and resilience.

The most visible characteristic of the system, its performance, has always been the weak point for persistent systems. This is usually attributed to stability and resilience mechanisms involving slow disk I/O. It is conceivable that the real reasons are buried deeper in the design of the system. One example is the definition of the basic persistent object which has a serious impact on all other aspects of the system design, including data sharing, process management, distribution, stability, etc.

More research and experimental work is needed before these issues are better understood. Practical solutions exist but they offer no justification for the approach adopted in the design or analysis of alternative solutions. The eventual success of persistence as a new computing paradigm depends on this work.

References


