Grasshopper: An orthogonally persistent operating system

[†]Alan Dearle, [*]Rex di Bona, [*]James Farrow, [*]Frans Henskens,
[*]Anders Lindström, [*]John Rosenberg, [†]Francis Vaughan

[†]Department of Computer Science
University of Adelaide
S.A., 5001, Australia
{al,francis}@cs.adelaide.edu.au

[*]Department of Computer Science
University of Sydney
N.S.W., 2006, Australia
{rex,matty,frans,anders,johnr}@cs.su.oz.au

Abstract

For ten years researchers have been attempting to construct programming language systems that support orthogonal persistence above conventional operating systems. This approach has proven to be poor; researchers invariably construct a complete abstract machine above the operating system with resulting loss of efficiency. This paper describes a new approach, the construction of an operating system designed to support orthogonal persistence. The operating system, Grasshopper, relies upon three powerful and orthogonal abstractions: *containers*, *loci* and *capabilities*. Containers provide the only abstraction over storage, loci are the agents of change, and capabilities are the means of access and protection in the system. This paper describes these three fundamental abstractions of Grasshopper, their rationale and how they are used.

1. Introduction

The aim of the Grasshopper project is to construct an operating system that supports orthogonal persistence [3]. This begs two questions, what is orthogonal persistence and why does it require support from an operating system?

The two basic principles behind orthogonal persistence are:

- that any object may persist (exist) for as long, or as short, a period as the object is required, and
- that objects may be manipulated in the same manner regardless of their longevity.

The requirements of a system which supports orthogonal persistence can be summarised as follows.

- Uniform treatment of data structures:

  Conventional programming systems require the programmer to translate data resident in virtual memory into a format suitable for long term storage. For example, graph structures must be flattened when they are mapped onto files or relations; this activity is both complex and error prone. In persistent systems, the programmer is not required to perform this mapping since data of any type with arbitrary longevity is supported by the system.

- Location independence:

  To achieve location independence, data must be accessed in a uniform manner, regardless of the location of that data. This principle is the cornerstone of virtual memory – the programmer does not have to worry if data is in RAM or on disk; the data is accessed in a uniform manner. In distributed persistent systems, location independence is extended to the entire computing environment by permitting data resident on other machines to be addressed in the same manner as local data [14, 15, 19, 40, 41]. This approach is also followed in distributed shared memory systems [36].

- Data resilience:

  All systems containing long-lived data must provide a degree of resilience against failure. In conventional operating systems, tools such as *fsck* in Unix permit the repair of long lived data (the file system) after a system crash. Persistent systems must also prevent the data stored in them becoming corrupt should a failure occur. However, the problem of resilience is more acute with persistent systems. In a conventional file system each file is essentially an independent object, and the loss of a single file following a system crash does not threaten the integrity of the overall system. In a persistent system, there may be arbitrary cross references between objects and the loss of a single object can be catastrophic. In addition, since one of the goals of persistence is to abstract over storage, resilience mechanisms should not be visible at the user level. In this sense the problem of recovery within a persistent system is more closely related to recovery in database systems [2].

- Protection of data:

  Persistent systems provide a large persistent store in which all data resides and against which all processes execute. A process may only access data for which it holds access permission. Failure by the operating system to provide a protection mechanism could result in erroneous processes corrupting data owned by other users. Therefore a protection mechanism must be provided to protect data from accidental or malicious misuse. In persistent systems this is typically provided via the programming language type system [27], through data encapsulation [25], using capabilities [12], or by a combination of these techniques.

To date, most persistent systems, with a few exceptions [7, 10, 32], have been constructed above conventional operating systems. Implementors of persistent languages are invariably forced to construct an abstract machine above the operating system, since the components of a persistent system are different in nature to the components of a conventional operating system. For example, in [37], Tanenbaum lists the four major components of an operating system as being memory management, file system, input-output and process management. In persistent systems, the file system and memory management components are unified. Some persistent systems require that the state of a process persists. This is not easily supported using conventional operating systems in which all processes are transitory.

The principal requirements of an operating system that supports orthogonal persistence may be summarised as follows [39]:

i. Support for persistent objects as the basic abstraction: persistent objects consist of data and relationships with other persistent objects. The system must therefore provide a mechanism for supporting the creation and maintenance of these objects and relationships.

ii. The system must reliably and transparently manage the transition between long and short term memory.

iii. Processes should be persistent.

iv. Control over access to objects: as described above, any operating system supporting persistence must provide some protection mechanism.

2. Grasshopper

Grasshopper is an operating system that provides support for orthogonal persistence. It is intended to run on a conventional hardware base, which has constrained the design of the system. A conventional hardware platform implies the lack of sophisticated features for the control of memory other than page-based virtual memory. Hence all notions of access control and structuring in Grasshopper are based on page-oriented virtual memory.

Grasshopper relies upon three powerful and orthogonal abstractions: *containers*, *loci* and *capabilities*. Containers provide the only abstraction over storage, loci are the agents of change, and capabilities are the means of access and protection in the system.

Conceptually, loci execute within a single container, their *host container*. Containers are not virtual address spaces. They may be of any size, including larger than the virtual address range supported by the hardware. The data stored in a container is supplied by a *manager*. Managers are responsible for maintaining a consistent and recoverable stable copy of the data represented by the container. As such, they are vital to the removal of the distinction between persistent and volatile storage, and hence a cornerstone of the persistent architecture.

This paper describes these three fundamental abstractions of Grasshopper, their rationale and how they are used. First, in Section 3, containers are described, followed, in Section 4 by issues relating to the sharing of data. Section 5 describes managers and how they operate. Sections 6 and 7 deal with loci and capabilities. Section 8 discusses issues related to the support for persistence in more detail. The paper concludes with an example of the use of Grasshopper.

3. Containers

In systems which support orthogonal persistence the programmer does not perceive any differences between data held in RAM and that on backing store. This idea leads naturally to a model in which there is a single abstraction over all storage. A major issue that arises is the addressing model supported by the system for this abstraction. There appear to be three basic models:

1. the single flat address space model,
2. the single partitioned address space model and
3. the fully partitioned address space model.

Models 1 and 3 represent opposite ends of the design spectrum where as model 2 is a hybrid architecture. These models are described in the following sections.

3.1 Single flat address space

In the first model all data resides in a single flat address space, this is the model adopted by the Napier88 persistent programming system [28]. The construction of very large stores using this technique was not feasible on conventional architectures until recently due to address size limitations. However, the advent of machines such as the DEC Alpha [35] and the MIPS R4000 [17] which (logically) support a 64 bit address has created renewed interest in this approach. A number of research groups have suggested that this is an appropriate direction for modern operating systems [20]. Such an approach is tempting since it fits in well with the goals of orthogonal persistence, i.e. to abstract over all physical attributes of data. However, there are some difficulties:

i. Most persistent systems rely upon a checkpointing mechanism to establish a consistent state on stable storage such as disk. If the operating system supports a single massive address space, the stabilisation mechanism must either capture the entire state of this store at a checkpoint, or record the dependencies between processes and data in the store and checkpoint dependent entities together. Even using incremental techniques, the first option could take a considerable amount of time due to I/O bandwidth.

ii. If a single address space is shared by all processes, the ability to protect separate areas of the address space must be provided. Whilst protection systems have been designed for large address spaces [20], they are still only experimental designs.

iii. The resulting store would be huge and the management of large stores is difficult. In particular allocation of free space, garbage collection, unique naming of new objects and the construction of appropriate navigation tools are all more difficult in large flat stores. These and other difficulties are discussed in [29].

3.2 Single partitioned address space

In the second model the notion of a large address space in which all objects reside is retained. However, this address space is partitioned into semi-independent regions. Each of these regions contains a logically related set of data and the model is optimised on the assumption that there will be few inter-region references. Such an approach was used for PS-algol [5] and is the basis of the Monads architecture [34]. Providing that control can be retained over the inter-region references it is possible to garbage collect and checkpoint regions (or at least limited sets of regions) independently, alleviating problems (i) and (iii) above [6, 33]. In addition, the partitioning provides a convenient mechanism for the generation of unique object names [14].

The major problem that remains with this approach is the issue of protection. It is necessary to restrict the set of addresses which can be generated by a process. One possiblity is to provide special-purpose hardware to support protection in a partitioned store and an implementation of such an architecture has been described previously [16, 32, 38, 43]. However, conventional architectures provide only page-based protection and therefore protection mechanisms similar to those proposed for flat stores must be employed.

3.3 Fully partitioned address space

In the third model the store is fully partitioned. Each region is independent and there is no global address space. Regions have names, however these may only be used to operate on regions as a whole. At any time a process is executing within a single region and may only access the data accessible from that region. The use of multiple independent regions has many advantages. Firstly, regions permit the logical grouping of related data which may improve performance in terms of disk access time, garbage collection and checkpointing overheads. Secondly, partitioning may also provide a level of protection, where required, between different uses of the store. For example, in a multilingual environment it is necessary to partition the store in order to ensure type security.

Grasshopper adopts this third approach by implementing regions called *containers*. Containers are the only storage abstraction provided by Grasshopper; they are persistent entities which replace both address spaces and file systems. In most operating systems, the notion of a virtual address space is associated with an ephemeral entity, a process, which accesses data within that address space. In contrast, containers and loci are orthogonal concepts. A Grasshopper system consists of a number of containers which may have loci executing within them. At any moment in time, a locus can only address the data visible in the container in which it is executing. Of course, there must be facilities which allow the transfer of data between containers. The mechanisms provided in Grasshopper are mapping and invocation and these are described in the following sections.

4. Container mappings

The purpose of container mapping is to allow data to be shared between containers. This is achieved by allowing data in a region of one container to appear in another container. In its simplest form, this mechanism provides shared memory and shared libraries similar to that provided by conventional operating systems. However, conventional operating systems restrict the mapping of memory to a single level. Both VMS [23] and variants of Unix (such as Sun-OS) provide the ability to share memory segments between process address spaces, and a separate ability to map from disk storage into a process address space. Several other systems [8, 9][26] provide the notion of a *memory object*, which provides an abstraction of data. In these systems, memory objects can be mapped into a process address space, however memory objects and processes are separate abstractions. It is therefore impossible to directly address a memory object, or to compose a memory object from other memory objects.

By contrast, the single abstraction over data provided by Grasshopper may be arbitrarily recursively composed. Since any container can have another mapped onto it, it is possible to construct a hierarchy of container mappings as shown in Figure 1. The hierarchy of container mappings form a directed acyclic graph. The restriction that mappings cannot contain circular dependencies is imposed to ensure that one container is always ultimately responsible for the data. In Figure 1, container *C2* is mapped onto container *C1* at location *a1*. In turn, *C2* has regions of containers *C3* and *C4* mapped onto it. The data from *C3* is visible in *C1* at address *a3*, which is equal to *a1* + *a2*.

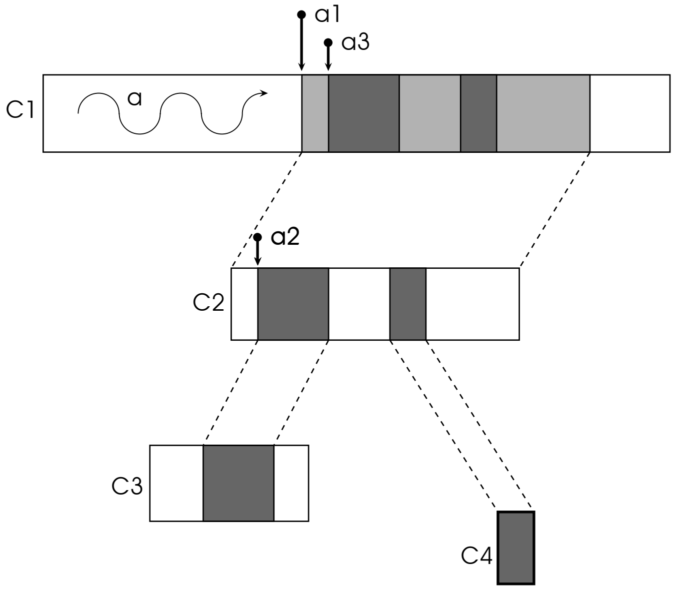

Figure 1: A container mapping hierarchy

Loci perceive the address space of their host container. Therefore all loci executing within a container share the same address space. However, a locus may require private data, which is visible to it, yet invisible to other loci that inhabit the same container. To satisfy this need, Grasshopper provides the notion of an invocation mapping.

Invocation mappings are local to the locus invocation in which they are created and take precedence over host container mappings. This allows, for example, each locus to have its own stack space with the stacks of all loci occupying the same address range within their host containers. This is shown in Figure 2 below. This technique both simplifies multi-threaded programming and provides a useful security mechanism that is unavailable with conventional addressing mechanisms.

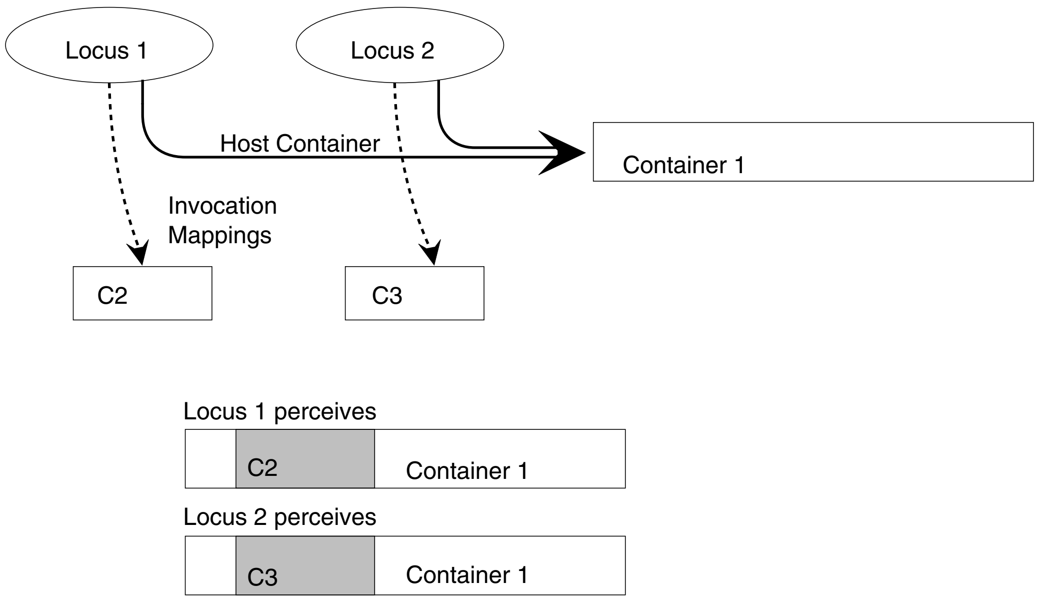

Figure 2: Loci with private stack space

5. Managers

Thus far we have described how all data storage in Grasshopper is provided by containers. However, we have not described how containers are populated with data. When data in a container is first accessed, the kernel is required to provide the concrete data that the container represents. A locus executing within a container accesses the data stored in it using container addresses. The container address of a word of data is its offset relative to the start of the container in which it is accessed. Managers are responsible for providing the required data to the kernel and are also responsible for maintaining the data when it is not RAM resident. Rather than being part of the kernel, managers are ordinary programs which reside and execute within their own containers; their state is therefore resilient. The concept of a manager is similar to the Mach external pager [31] which has been successfully used to implement a coherent distributed persistent address space [19]. In common with Mach, managers are responsible for:

- provision of the pages of data stored in the container,
- responding to access faults, and
- receiving data removed from physical memory by the kernel.

In addition, Grasshopper managers have the following responsibilities:

- implementation of a stability algorithm for the container [6, 21, 26, 33, 41], i.e. they maintain the integrity and resilience of data, and
- maintenance of coherence in the case of distributed access to the container [15, 24, 30].

A manager is invoked whenever the kernel detects a memory access fault to data stored in the container it manages. Making data accessible in a container takes place in two steps:

i. the manager associated with a particular address range must be identified, and,

ii. the appropriate manager is requested to supply the data.

The kernel is responsible for identifying which manager should be requested to supply data. This is achieved by traversing the container mapping hierarchy. Once the correct manager has been identified, the kernel requests this manager to supply the data. The manager must deliver the requested data to the kernel, which then arranges the hardware translation tables in such a way that the data is visible at an appropriate address in the container.

In Grasshopper, the manager is the only mechanism by which data migrates from stable to volatile storage. This is in contrast to conventional operating systems in which the usual abstraction of stable storage is the file system. Grasshopper has no file system in the conventional sense.

Managers are responsible for maintaining a resilient copy of the data in a container on stable media. It is only within a manager that the distinction between persistent and ephemeral data is apparent. Managers can provide resilient persistent storage using whatever mechanism is appropriate to the type of data contained in the managed container. Since managers are responsible for the storage of data on both stable media and in RAM they are free to store that data in any way they see fit. An important class of managers are those that store data on stable media in some form other than that viewed by a locus in a container; we term these manipulative managers. Some examples of manipulative managers are:

i. swizzling managers,

ii. encrypting managers, and,

iii. compressing managers.

Swizzling managers are particularly interesting in that they permit the use of containers that are larger than the address space supported by the host hardware. A locus executing within a large container will generate addresses constrained by the machine architecture. Access faults will be delivered by the kernel to the manager associated with a container. A swizzling manager will provide a page of data in which addresses that refer to objects anywhere within the container are replaced with shorter addresses (ones within the machine's address range) which, when dereferenced will be used by the swizzling manager to provide the correct data [4, 39, 42].

It is possible for a locus to execute in a container whose manager provides a one-to-one mapping between data in virtual memory and data on disk. This is the abstraction delivered by demand paged virtual memory and memory mapped files in conventional operating systems. Most managers will not operate in this manner since this does not adequately support orthogonal persistence.

6. Loci

In Grasshopper, loci are the abstraction over execution. In its simplest form, a locus is simply the contents of the registers of the machine on which it is executing. Like containers, a locus is maintained by the Grasshopper kernel and is inherently persistent. Making the locus persistent is a departure from other operating system designs and frees the programmer from much complexity.

A locus is associated with a container, its host container. The locus perceives the host container's contents within its own address space. Virtual addresses generated by the locus map directly onto addresses within the host container. A container comprising program code, mutable data and a locus forms a basic running program. Loci are an orthogonal abstraction to containers. Any number of loci may execute within a given container; this allows Grasshopper to support multi-threaded programming paradigms.

6.1 Inter-Container Communication

An operating system is largely responsible for the control and management of two entities: objects, which contain data (containers); and processes (loci), the active elements which manipulate these objects. One of the most important considerations in the design of an operating system is the model of interaction between these entities. There are two principal models of computation called *message oriented* and *procedure oriented* [22]. In the message oriented model, processes are directly associated with objects and communication is achieved through the use of messages. By contrast, the procedure oriented model provides processes that move between objects. Processes access objects by invoking them; the invoke operation causes a (possibly remote) procedure call to code within the object. By executing this code the process may access data stored within the object. The message oriented model cannot be used to efficiently simulate any other computational models. The procedure oriented model is more flexible; for instance, it can easily simulate the message oriented model by associating a message passing process with every object. For this reason, Grasshopper uses the procedure oriented model in which a locus may invoke a container thereby changing its host container.

Any container may include as one of its attributes a single entry point known as an *invocation point*. When a locus invokes a container, it begins executing code at the invocation point. The single invocation point is important for security; it is the invoked container that controls the execution of the invoking locus by providing the code that will be executed. Invocation mappings are not retained across invocation since this could be used to avoid executing the provided code and thus subvert security.

The invoking locus provides a fixed size parameter block to the kernel mediated invoke operation. This parameter block is made available to the code in the invoked container. For many invocations this parameter block will suffice. However, larger amounts of data may be passed via an intermediate container. This technique is illustrated in Section 9. Appropriate and arbitrarily sophisticated communication protocols may be built on top of this simple facility. Since a minimal parameter block is the only context that is transferred to the invoked container, invocation is therefore inherently low cost. In this respect, the invoke primitive is very similar to the message passing system used in the V-kernel [8].

A locus may invoke and return through many containers in a manner similar to conventional procedure call. The Grasshopper kernel maintains a call chain of invocations between containers. Implicitly each locus appears to be rooted in the container representing the kernel: when a locus returns to this point it is deleted. However some loci may never need to return to the container from which they were invoked. Such a locus may meander from container to container. In such circumstances, an invoke parameter allows the locus to inform the kernel that no return chain need be kept.

7. Capabilities

Capabilites provide an access control mechanism over containers and loci. For containers, access control is required over:

- container maps,
- the containers which may be invoked,
- the ability to set an invocation point,
- the containers which may be mapped and the access type available to mapped regions (read/write), and
- deletion of containers,

and for loci access control is required over:

- creation of invocation mappings,
- blocking and unblocking loci,
- management of exceptions and
- deletion of loci.

In conventional operating systems these access controls are usually provided by the file system interface. This is clearly inappropriate for Grasshopper. In several existing persistent systems protection is provided via the programming language type system [27] or through data encapsulation [25]. Grasshopper is intended to support multiple languages and therefore cannot rely solely on a type system.

The protection abstraction provided by Grasshopper is the *capability* [12]. Capabilities were first proposed by Dennis and Van Horn [11] and have been used in a variety of contexts as an access control mechanism [16, 32, 38, 43]**.** A capability consists of a unique name for an entity, a set of access rights related to that entity, and rights pertaining to the capability itself, in particular whether the capability can be copied. An operation can only be performed if a valid capability for that operation is presented. There are two important points about capabilities from which they derive their power: the name of the entity is unique and capabilities cannot be forged or arbitrarily modified. Capabilities can only be created and modified by the system in a controlled manner.

There are well-known techniques for achieving the above requirements. Unique names may be generated by using a structured naming technique where each machine is given a unique name and each entity created on that machine has a unique name. Such a technique is described in [15]. Protection of capabilities can be achieved in one of three ways:

tagging:	in which extra bits are used by the hardware to indicate memory regions representing capabilities and to restrict access to these regions,
segregation:	in which capabilities are stored in a protected area of memory,
passwords:	in which a key embedded in a sparse address space is stored with the entity and a matching key must be presented to gain access to that entity.

The merits of each of these techniques have been well discussed in the literature [1, 13, 18, 38]. Given that Grasshopper is to be implemented on conventional hardware, tagging is not an option. Segregation is used in Grasshopper since it avoids the problems associated with knowing when to garbage collect unreferenced entities. This occurs with password capabilities since a user may keep a copy of the capability somewhere outside of the kernel's control. Since the kernel cannot know how many (if any) of these externally recorded capabilities may be in existence it cannot perform garbage collection except on entities it is specifically told to destroy. Using segregated capabilities allows garbage collection to be performed in association with explicit destruction of entities by a locus. When the reference count on a capability falls to zero, i.e., there are no more extant references to the corresponding entity, the entity may be deleted.

In Grasshopper, every container and locus can have an associated list of capabilities. Operations are provided for copying capabilites and for adding and removing them to and from lists. At any time, a locus has access to:

i. all the capabilities in its own list,

ii. all capabilities in its host container's list,

iii. all the capabilities associated with containers mapped into its host container and

iv. all the capabilities associated with containers in the locus' invocation mappings.

Programs can refer to capabilities by specifying the name of an entity and an index into its capability list. Grasshopper checks that the name represents an entity which is currently in scope (i.e. whose capability list is visible) and that the index is valid. An appropriate capability must be presented for most operations involving the manipulation of entities, such as invocation and mapping.

A number advantages arise from the use of capabilities for access and protection:

i. Distribution is completely transparent: A locus wishing to invoke a container simply presents the capability. The capability uniquely identifies the container and its physical location is irrelevant.

ii. The system does not force any particular protection structure upon users. It is possible to construct hierarchical protection or more general policies using arbitrary naming mechanisms which map some representation of a name onto a capability.

iii. It is possible to create restricted views of objects. For example, two different capabilities for a container could be created, one of which allowed the container to be mapped, while the other only allows it to be invoked.

iv. It is possible to revoke access. A holder of a capability with appropriate access rights can invalidate other capabilities for the same entity.

8. Persistence

Containers and their associated managers provide the abstraction of persistent data. Managers are responsible for maintaining a consistent and recoverable stable copy of the data represented by the container. As part of its interface, each manager must provide a stabilise operation [6, 33]. Stabilisation involves creating a consistent copy of the data on a stable medium.

Managers alone are not able to maintain a system-wide consistent state. For example, consider the case where two containers, A and B, both provide data which is used and modified by a single program. The manager for container A stabilises the state of A, and execution of the main program continues. At a later time, container B is stabilised. This does not result in a consistent view of data from the point of view of the executing program, since after a system crash the recovered state of the two containers are inconsistent relative to one another.

The simplest approach to global consistency is to enforce system wide simultaneous stabilisation in which the kernel blocks all executing loci and requests each manager to stabilise. The disadvantage of this approach is that the entire system freezes whilst the stabilise occurs. Poor performance could also result since it would cause a dramatic increase in I/O activity which could swamp the available bandwidth of disks and controllers.

An alternative approach is to determine which containers and loci are causally dependent upon each other's state and only force these to stabilise together, leaving the rest of the system to run. Such inter-dependent units are termed *associates* [40]. The kernel may determine which containers and loci are inter-dependent by annotating the container mapping graph with a history of invocations and dependencies on kernel data. The internal state of kernel data structures also forms part of the state of a user program. For example, the granting of capabilities to loci must be recorded. The causal dependencies must therefore be extended to include kernel data. Thus a complete, consistent and recoverable representation of a subsection of the system can be produced.

When a consistent subset of the system has been determined, the kernel proceeds with a two phase commit protocol, requesting the appropriate container managers to stabilise the data in their charge. When all user data is stable, the kernel will proceed to stabilise its own state, and finally, as an atomic action, will commit a descriptor block that describes the new state to stable medium.

In this way, the kernel is part of the persistent environment, thereby extending the concept of an operating system instance. A Grasshopper kernel persists even when the host machine is not operating. Conventional operating systems rebuild the operating system from scratch each time they are bootstrapped. In Grasshopper, the entire kernel, operating system and user state persists. After an initial bootstrap, an entire self-consistent state is loaded and continues execution.

9. An Example

To illustrate the use of the abstractions provided by Grasshopper, consider the following example. A locus executing in a container (the *client container*) wishes to make use of a service provided by another container (the *server container*). In this example, we assume that the server is trusted to perform the requested service but neither the client nor the server trust the other to allow direct access to their state through mapping. To achieve an interface between the two containers which permits the server to safely provide services to the client, the client and server elect to use a common communications interface. Since the communications library is trusted, it may be safely mapped into both containers, and the interface code directly executed. This is shown in Figure 3 below.

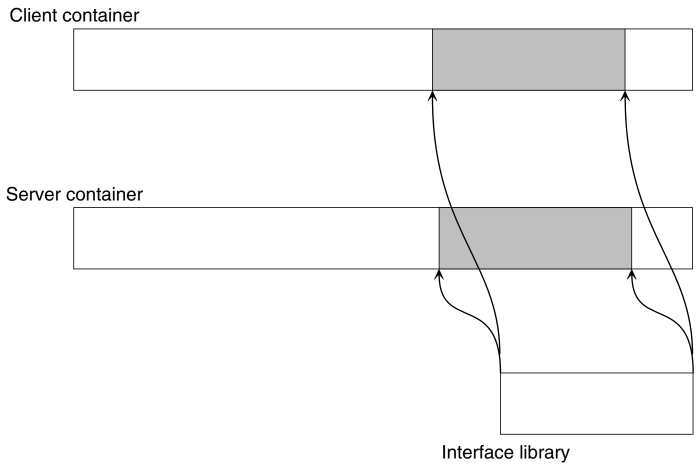

Figure 3: Client and server containers map the interface library.

The interface library abstracts over the complexity of inter-container communication. This is achieved by presenting a procedural abstraction of the server's functionality to the client. The actual inter-container communication is performed by an invoke system call between the interface library code within the client container and the same code within the server container. To use the service, the client uses an appropriate routine from the interface library. This routine assembles an appropriate parameter block and then invokes the server, as shown in Figure 4.

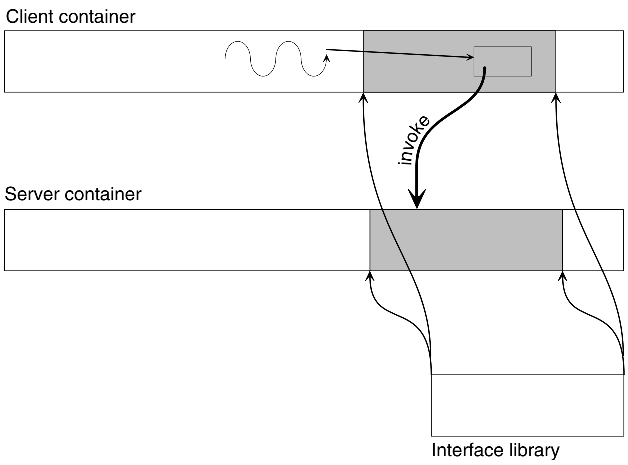

Figure 4: A locus calls an interface function which invokes the server container.

If the service requested requires that a large amount of data be transferred between the client and server, it can utilise a trusted intermediary container as follows. Firstly, the interface library in the client creates a new container, *P*. Once created, *P* is mapped into the client container at a free address. Finally, a capability for *P* is inserted into the set of capabilities owned by the running locus. The locus now has access to *P* wherever it may be running. Next, the interface library invokes a function that maps *P* into the server container. This is possible since the locus carries a capability for *P*. As shown in Figure 5, subsequent communication takes place using other functions in the shared interface library. These functions may exchange data using *P* as a trusted intermediary without compromising either the client or server containers.

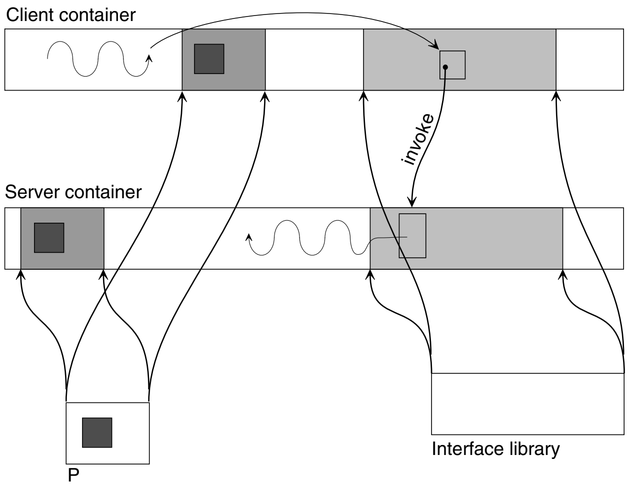

Figure 5: Using an intermediate container to support the transfer of large amounts of data.

10. Conclusions

In this paper the initial design of The Grasshopper Persistent Operating System has been described. Grasshopper satisfies the four principle requirements of orthogonal persistence namely:

i. support for persistent objects as the basic abstraction,

ii. the reliable and transparent transition of data between long and short term memory,

iii. persistent processes and

iv. control over access to objects.

This is achieved through the provision of three powerful and orthogonal abstractions namely *containers*, *loci* and *capabilities*. Containers provide the only abstraction over storage, loci are the agents of change, and capabilities are the means of access and protection in the system. These abstractions are supported by a fourth mechanism, the manager, which is responsible for data held in a container.

Based on our experience of constructing persistent systems, we believe that these abstractions will provide an ideal base for persistent programming languages. At the moment we cannot prove this assertion since the Grasshopper system is still under construction. Initial implementation experiments were performed on a Sun 3/60. Implementation is currently proceeding on a DEC Alpha platform.

Acknowledgements

We would like to thank Karen Wyrwas, Alex Farkas, Stephen Norris, Fred Brown and David Hulse for comments on earlier versions of this paper.

This work is supported by a grant number A49130439 from the Australian Research Council and by an equipment donation from Digital Equipment Corporation.

References

1. Anderson, M., Pose, R. and Wallace, C. S. "A Password-Capability System", *The Computer Journal*, vol 29, 1, pp.1-8, 1986.
2. Astrahan, M. M. "System R: Relational Approach to Database Management", *ACM Transactions on Database Systems*, vol 1, 2, pp.97-137, 1976.
3. Atkinson, M. P., Bailey, P. J., Chisholm, K. J., Cockshott, W. P. and Morrison, R. "An Approach to Persistent Programming", *The Computer Journal*, vol 26, 4, pp.360 - 365, 1983.
4. Atkinson, M. P., Bailey, P. J., Cockshott, W. P., Chisholm, K. J. and Morrison, R. "POMS: A Persistent Object Management System", *Software Practice and Experience*, vol 14, 1, pp.49-71, 1984.
5. Atkinson, M. P., Chisholm, K. J. and Cockshott, W. P. "PS-algol: An Algol with a Persistent Heap", *ACM SIGPLAN Notices*, vol 17, 7, pp.24-31, 1981.
6. Brown, A. L. "Persistent Object Stores", Ph.D Thesis, 1988.
7. Campbell, R. H., Johnston, G. M. and Russo, V. F. "Choices (Class Hierarchical Open Interface for Custom Embedded Systems", *ACM Operating Systems Review*, vol 21, 3, pp.9-17, 1987.
8. Cheriton, D. R. "The V Kernel: A Software Base for Distributed Systems", *Software*, vol 1, 2, pp.91-42, 1984.
9. Chorus-Systems "Overview of the CHORUS Distributed Operating Systems", *Computer Systems - The Journal of the Usenix Association*, Vol 1 No 4., 1990.
10. Dasgupta, p., LeBlanc, R., Mustaque, A. and Umakishore, R. "The Clouds Distributed Operating System", Technical Report Thesis, 88/25, 1988.
11. Dennis, J. B. and Van Horn, E. C. "Programming Semantics for Multiprogrammed Computations", *Communications of the A.C.M.*, vol 9, 3, pp.143-145, 1966.
12. Fabry, R. S. "Capability-Based Addressing", *CACM*, vol 17, 7, pp.403-412, 1974.
13. Gehringer, E. F. and Keedy, J. L. "Tagged Architecture: How Compelling are its Advantages", *Twelfth International Symposium on Computer Architecture*, pp.162-170, 1985.
14. Henskens, F. A. "Addressing moved modules in a capability based distributed shared memory", *25th Hawaii International Conference on System Sciences*, Kauai, Hawaii, pp.769-778, 1992.
15. Henskens, F. A., Rosenberg, J. and Keedy, J. L. "A Capability-based Distributed Shared Memory", *Proceedings of the 14th Australian Computer Science Conference*, Sydney, Australia, pp.29.1-29.12, 1991.
16. Hurst, A. J. and Sajeev, A. S. M. "Programming Persistence in χ", *I.E.E.E. Computer*, vol 25, 9, pp.57-66, 1992.
17. Kane, G. and Heinrich, J. "MIPS RISC Architecture", 1992.
18. Keedy, J. L. "An Implementation of Capabilities without a Central Mapping Table", *Proc. 17th Hawaii International Conference on System Sciences*, pp.180-185, 1984.
19. Koch, B., Schunke, T., Dearle, A., Vaughan, F., Marlin, C., Fazakerley, R. and Barter, C. "Cache Coherence and Storage Management in a Persistent Object System", *Proceedings, The Fourth International Workshop on Persistent Object Systems*, Marthas Vineyard, pp.99-109, 1990.

20. Koldinger, E., Chase, J. and Eggers, S. “Architectural Support for Single Address Space Operating Systems”, *Architectural Support for Programming Languages and Operating Systems*, Boston, Mass, pp.175-197, 1992.
21. Lampson, B. *Distributed Systems Architectures and Implementation*, vol 105, pp.250-265,
22. Lauer, H. C. and Needham, R. M. “On the Duality of Operating System Structures”, *Operating Systems Review*, vol 13, 2, pp.3-19, 1979.
23. Levy, H. M. and Lipman, P. H. “Virtual Memory Management in the VAX/VMS Operating System”, *Computer*, vol 15, 3, pp.35-41, 1982.
24. Li, K. “Shared Virtual Memory on Loosely Coupled Multiprocessors”, Ph.D. Thesis, 1986.
25. Liskov, B. H. and Zilles, S. N. “Programming with Abstract Data Types”, *SIGPLAN Notices*, vol 9, 4, 1974.
26. Lorie, R. A. “Physical Integrity in a Large Segmented Database”, *Association for Computing Machinery Transactions on Database Systems*, vol 2, 1, pp.91-104, 1977.
27. Morrison, R., Brown, A. L., Connor, R. C. H., Cutts, Q. I., Kirby, G. N. C., Dearle, A., Rosenberg, J. and Stemple, D. “Protection in Persistent Object Systems”, *Security and Persistence*, pp.48-66, 1990.
28. Morrison, R., Brown, A. L., Connor, R. C. H. and Dearle, A. “The Napier88 Reference Manual”, PPRR-77-89, 1989.
29. Moss, J. E. B. “Addressing Large Distributed Collections of Persistent Objects: The Mneme Project's Approach”, *2nd International Workshop on Database Programming Languages*, Salishan, Oregon, San Mateo, California, pp.358-374, 1989.
30. Philipson, L., Nilsson, B. and Breidegard, B. “A Communication Structure for a Multiprocessor Computer with Distributed Global Memory”, *10th International Symposium on Computer Architecture*, Stockholm, pp.334-340, 1983.
31. Rashid, R., Tevanian, A., Young, M., Golub, D., Baron, R., Black, D., Bolosky, W. and Chew, J. “Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures”, *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II)*, Palo Alto, pp.31-39, 1987.
32. Rosenberg, J. and Abramson, D. A. “MONADS-PC: A Capability Based Workstation to Support Software Engineering”, *18th Hawaii International Conference on System Sciences*, pp.515-522, 1985.
33. Rosenberg, J., Henskens, F. A., Brown, A. L., Morrison, R. and Munro, D. “Stability in a Persistent Store Based on a Large Virtual Memory”, *Proceedings of the International Workshop on Architectural Support for Security and Persistence of Information*, Bremen, West Germany, pp.229-245, 1990.
34. Rosenberg, J., Keedy, J. L. and Abramson, D. “Addressing Mechanisms for Large Virtual Memories”, *The Computer Journal*, vol 35, 4, pp.369-375, 1992.
35. Sites, R. L. “Alpha Architecture Reference Manual”, 1992.
36. Tam, M., Smith, J. M. and Farber, D. J. “A Taxonomy-based Comparison of Several Distributed Shared Memory Systems”, *Operating Systems Review*, vol 24, 3, pp.40-67, 1990.
37. Tanenbaum, A. S. “Operating Systems: Design and Implementation”, *International Editions*, 0-13-637331-3, 1987.

38. Tanenbaum, A. S. "Experiences with the Amoeba Distributed System", *Communications ACM*, vol 33, 12, pp.46-63, 1990.

39. Vaughan, F. and Dearle, A. "Supporting Large Persistent Stores Using Conventional Hardware", *5th International Workshop on Persistent Object Systems*, vol (to appear), San Miniato, Italy, 1992.

40. Vaughan, F., Schunke, T., Koch, B., Dearle, A., Marlin, C. and Barter, C. "A Persistent Distributed Architecture Supported by the Mach Operating System", *Proceedings of the 1st USENIX Conference on the Mach Operating System*, Burlington, Vermont, pp.123-140, 1990.

41. Vaughan, F., Schunke, T., Koch, B., Dearle, A., Marlin, C. and Barter, C. "Casper: A Cached Architecture Supporting Persistence", *Computing Systems*, vol 5, 3, California, 1992.

42. Wilson, P. "Pointer Swizzling at Page Fault Time: Efficiently Supporting Huge Address Spaces on Standard Hardware", *ACM Computer Architecture News*, June, pp.6-13, 1991.

43. Wulf, W. A., Levin, R. and Harbinson, S. P. "C.mmp/Hydra: An Experimental Computer System", NewYork, 1981.