Causality Considerations in Distributed, Persistent Operating Systems

*Francis Vaughan, *Alan Dearle,*Jiannong Cao,
†Rex di Bona, †J. Matthew Farrow, †Frans Henskens,
†Anders Lindström, †John Rosenberg.

*Department of Computer Science
University of Adelaide, S.A., 5005
Australia

{francis, al, jiannong}@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

In this paper we describe a new persistent distributed operating system. The Grasshopper system is designed to allow flexibility in the way in which persistence is provided. A key element of this flexibility is concerned with issues of global consistency. The Grasshopper kernels cooperate with user level entities in order to maintain and find globally consistent states using vector time. Further flexibility is provided by allowing the kernels to implement either eager or lazy consistency policies.

1 Introduction

The aim of the Grasshopper project is to construct an operating system that supports orthogonal persistence 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.

Any system containing long-lived data must provide some element of resilience against failure. The problem of resilience is especially acute in persistent systems; since they contain data of arbitrary longevity, they must prevent the data stored in them becoming corrupt in the event of a failure. 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. Since persistence systems abstract over storage mechanisms, resilience mechanisms should not be visible at the user level. Provision of this resilience is the central thread of this paper.

The other 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. 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 [9, 10, 12, 25, 26].

- Protection of data:

  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 [20], through data encapsulation [17], using capabilities [6], or by a combination of these techniques.

The Grasshopper project seeks to provide a mechanism that provides resilience as an intrinsic attribute of a distributed, orthogonally persistent operating system.

1.1. Grasshopper

The Grasshopper operating system 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.

Containers are the only storage abstraction provided by Grasshopper; they are persistent entities which replace both address spaces and file systems. In contrast to most operating systems, containers and loci are orthogonal concepts. A Grasshopper system consists of a number of containers some of which may have loci executing within them.

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. Loci are maintained by the Grasshopper kernel and are 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. Thus, 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. Loci are not permanently associated with any container, a locus may move from one host container to another through the *invoke* mechanism. Invocation is the basic mechanism of communication in Grasshopper.

To order for a locus to invoke another container, it must hold an appropriate capability for that container and that container must have an *invocation point*. An invocation point is a specified address within a container at which an invoking locus will begin execution. When a locus invokes another container it always starts executing the code at the entry point supplied by that container.

The Grasshopper kernel maintains an optional invocation call stack which permits a locus to return to its caller. The locus may provide a parameter block which the kernel makes available to the locus when it has transferred to the new host container.

All operations provided by the system are modelled as an invocation of the appropriate container. Kernel level operations are modelled as invocation of a distinguished container that represents the kernel.

The Grasshopper system allows data to be shared using a mechanism known as *container mapping*. Mapping allows 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.

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 which 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.

The protection abstraction provided by Grasshopper is the *capability* [6]. 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. Further discussion of capabilities is outside the scope of this paper. Capabilities in Grasshopper are described in more detail in [5]

Managers are the mechanism by which persistence of user data is provided, they are responsible for delivering required data to the kernel and 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. Managers are responsible for:

- provision of the pages of data stored in the container,
- responding to access faults,
- receiving data removed from physical memory by the kernel,
- implementation of a stability algorithm for the container [3, 14, 18, 22, 26], i.e. they maintain the integrity and resilience of data, and
- maintenance of coherence in the case of distributed access to the container [10, 15, 21].

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.

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. As at some time, the manager for container A might checkpoint the state of A, and execution of the main program continues. At a later time, container B is checkpointed. However, this does not result in a global consistent state; if the system were to be restarted, the program would view a situation which could never have arisen through correct execution.

Since containers A and B are causally connected, in order to maintain correctness, the system must ensure that their state mutually consistent. When a number of containers are causally inter-dependent in this way, one container reverting to an earlier state may cause a cascade of reversion in other containers. This is often termed the domino effect and may require reverting the collection to the start of computation, a time at which the collection is trivially consistent.

2. Stability and Causality

When distributed concurrent processes co-operate it is necessary to co-ordinate their activity in some manner. A central problem is the impossibility of providing universal time in a distributed system. Instead, the notion of *causality* is useful for reasoning about distributed computations. Causality can intuitively be defined in terms of some entity being in some way dependant upon the state of another. For this to happen, state information must be passed between entities.

More formally, causality may be captured using the *happened-before* relation [13]. Briefly, the happened before relation "$\rightarrow$" is the smallest relation such that:

If a and b are events in the same entity and a comes before b then a Æ b

If a is the sending of information from one entity and b is its receipt in another then a $\rightarrow$ b

If a $\rightarrow$ b and b $\rightarrow$ c then a $\rightarrow$ c

If a $\not\rightarrow$ b and b $\not\rightarrow$ a then a and b are said to be concurrent.

Causality in distributed systems is very similar to relativistic effects in the physical world caused by the finite speed of light. It is impossible for some action occurring at a distance to effect an observer until results of the action traverse the gap. Often the results of some event are represented as a *light cone* radiating out from the event. Similarly the domain of events that can effect a point may be presented by a *causal cone* which expands backwards in time. In Figure 1

only those actions that occur within the cone can affect the point represented at the apex of the cone. Action α occurs within the causal cone and its effect is visible at χ. However β lies outside the causal cone and therefore χ cannot be causally dependant upon it. In a similar manner the causal cone in a computational system may be modelled.

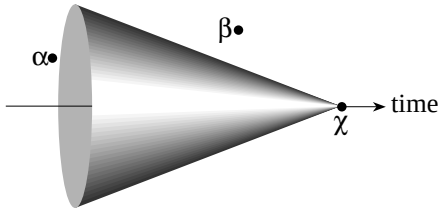

Figure 1. A causal cone in space time. a can effect c but not b.

Figure 2 represents a distributed computation in which each horizontal line represents a single node with time increasing to the right. Communication between nodes is represented by a directed arc between time lines. In Figure 2, node N3 at time χ is affected by an action α in N1, but action β in invisible to it. Thus χ is causally dependant upon α but not β.

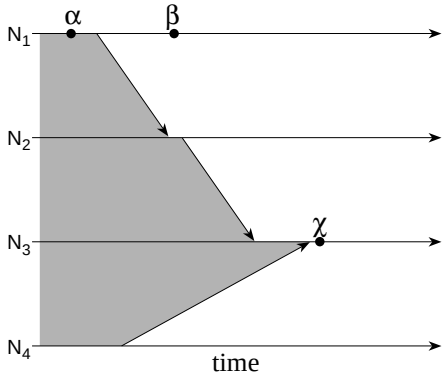

Figure 2. A causal cone in a distributed system. Event α can effect χ but not β.

An operating system that provides orthogonal persistence must provide some failure recovery mechanism. In this paper we shall assume that the system is fail stop. That is, if a failure occurs, that component will cease to operate. Upon recovery, the system must find some consistent system state from the separately checkpointed states of system components. The problem is how to generate and find a set of these committed states that form a consistent global state; such a subset is known as a *consistent cut*.

Upon recovery, a useful consistent cut is one that represents some possible correct system state. This need not represent the system as it actually existed at some moment in time, indeed it probably does not. It must, however, represent a possible state; that is one reachable through some correct execution of the system components.

Formally a consistent cut is a subset of the events which comprise the system such that for all events in the cut, if e is an element of the consistent cut and e' $\rightarrow$ e then e' is also an element of the cut [19].

Graphically, consistent cuts are easily conveyed as a line drawn downwards through a time diagram; dividing the diagram into two parts with the past on the left and the future on the right. A cut is consistent if no arrow (a communication) starts in the future and ends in the past, and thus no message is received before it is sent. The intersection of the cut line and a node's time line represents the time at which a snapshot of the state of the node was committed.

In order to maintain consistency, either outgoing messages must be recorded as part of the state of a node or a cut may not cross any message arrow. In either case, a message is never lost. Figure 3 shows four possible cuts in which α and γ are consistent cuts, δ is not, β is, only if messages in transit are recorded.

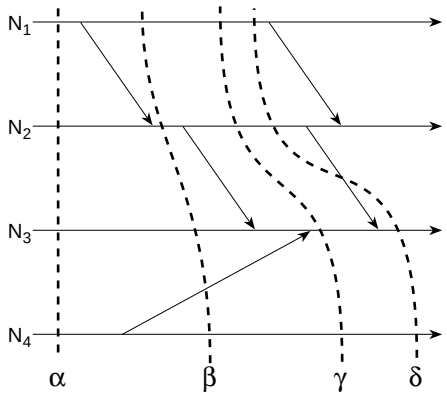

Figure 3. Consistent and inconsistent cuts.

The design of Grasshopper seeks to provide intrinsic support for orthogonal persistence. However, since it is an experimental system, it seeks to avoid imposing a single model of system stability upon its users. As is discussed in section 4 below, a wide spectrum of choices are available to the designer of the stability mechanism. These choices trade recovery performance against failure free performance and further choices over the externally perceived reliability of the system.

2.1. Lamport Time

Lamport [13] describes a method of providing a global clock by which causal relationships can be characterised. Each event is tagged with an integer and the *happened before* relationship is implemented by comparing the tags for different events. As shown in Figure 4, using Lamport time, each node maintains a counter, initially zero. Whenever a message is sent the counter is incremented and the message is tagged with the counter's value. Upon receipt of a message, the local counter is set to the maxima of the local counter and the counter in the message. Using Lamport time, events with the same time in different nodes are arbitrarily ordered using some secondary criteria such as node number.

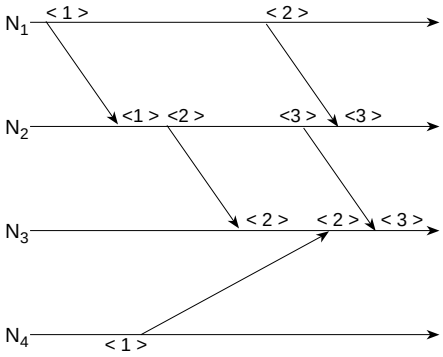

Figure 4. Lamport Time.

In a distributed system, message passing occurs concurrently in separate parts of the system; therefore the ordering of events is characterised by a partial ordering. However, Lamport time presents a single total ordering of events which is one of many possible correct total orderings. Extending Lamport time to express the partial ordering inherent in a distributed system leads to *vector time*.

2.2. Vector Time

Vector time has been described by a number of researchers [7, 23] and may be viewed as an extension of Lamport Time. Rather than keeping a single counter, each entity within the system maintains a time vector, in which each element represents knowledge about other entities within the system.

Vector time is maintained as follows; each vector has as many elements as there are entities within the system. $VT_i[j]$ denotes the j^{th} element of the time vector for entity i.

Each element of every vector in initially zero.

Whenever a message is sent to another entity, the sender increments the element of its vector corresponding to itself; that is $VT_i[i] = VT_i[i] + 1$ and this vector is transmitted with the message.

Upon receipt of a message, the receiving entity updates its own vector as follows. If any element of its own vector is greater than the corresponding element of the vector received, it is untouched. If an element of the received vector is greater than the corresponding element of the receiver's vector, the receiver updates that element to equal the received value.

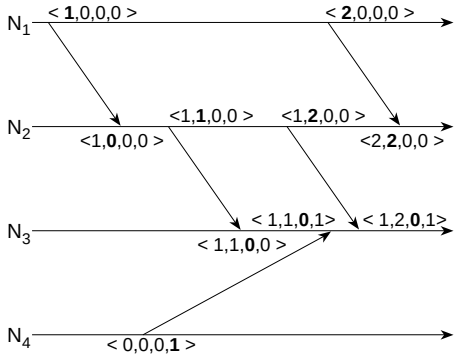

Figure 5. Vector Time.

Figure 5 shows a simple example of vector time. Using vector time, event α on node N_i happens before event β on node N_j iff,

$$VT_\alpha[i] < VT_\beta[i]$$

In effect this says that node i must have communicated with node j, perhaps via some intermediaries. In addition to the happened before relation, vector times also encompass the notion of concurrency. Two events α and β are concurrent if

$$\alpha \not\rightarrow \beta \text{ and } \beta \not\rightarrow \alpha$$

For example, the nodes labelled <1,0,0,0> and <0,0,0,1> are concurrent events. Unlike Lamport time, vector time captures the partial ordering of events within a system. Thus, vector time captures the *happens before* relationship and the notion concurrency in the system. It therefore contains all the information needed to formulate a consistent cut.

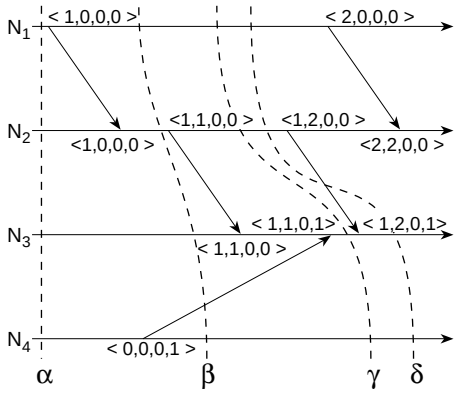

Figure 6: Consistent cuts and vector time

Vector time may be used to characterise consistent cuts. Johnson and Zwaenepeol [11] show a method which involves the construction of a *dependency matrix* which is a matrix whose rows correspond to the vector times of all nodes in a system. For example, the matrix for the cut δ in Figure 6 above is:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & \mathbf{1} & 0 & 0 \\ 1 & \mathbf{2} & 0 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

They show that a dependency matrix M represents a consistent cut iff,

$$\forall i,j \; M_{ij} \leq M_{ii}.$$

In other words, the elements in the column j must be less than or equal to the diagonal element. This says that no node i depends upon an event that has happened after the consistent cut was established. The cut denoted by δ is not consistent since entity three's value for entity two (2) is greater than entity two's own value (1). This represents the reception of a message by entity three without the corresponding message send taking place.

3. Maintaining Global Consistency

Many different methods for maintaining or finding a globally consistent state for persistent systems exist. These designs are guided by two orthogonal issues: the cost of failure and whether deterministic execution is possible.

Methods that take account of the cost of failure can be characterised as optimistic or pessimistic. Optimistic algorithms assume that failures occur with sufficient rarity, that it is better to place the bulk of the work needed for system recovery in the recovery phase, speeding up normal execution. Pessimistic systems place the main burden of work needed to maintain a recoverable state into the normal execution of the system. This may be because they take the view that failures are quite likely, or that the need to recover quickly from failures is sufficiently important. Systems which require the user view to be one of total reliability also place a greater burden on normal execution and can also be characterised as pessimistic.

Determinism of execution is important when systems recover through replay. In a system in which all the components execute in a deterministic manner almost arbitrary recovery may be performed. In principle, in a deterministic system, a program can be restarted from an arbitrary point and it will recover to the point at which the system failed.

Even if a deterministic program interacts with non-deterministic external agents, replay can still be used as a recovery mechanism. This may be achieved by logging all incoming messages to stable media and keeping a stable counter of outgoing messages. Upon system restart, the logged incoming messages are replayed and output is discarded until the number of output messages equals the stable counter.

However, if a program does not execute in a deterministic manner, it is not possible to use replay to recover from failure. The only recovery mechanism that may be employed is checkpointing. The Grasshopper system aims to support both paradigms.

The simplest approach to global consistency is to enforce system wide simultaneous stabilisation. The disadvantage of this approach is that the entire system freezes whilst the global checkpoint 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.

Mattern [19] describes an algorithm which allows a consistent stable state to be generated incrementally by tagging all outgoing messages after a stabilisation. On receipt of a tagged message a node must stabilise before it communicates with further nodes. Since this node is now stable, all messages sent by it are also tagged. This mechanism produces a consistent cut, but is intrusive and is complicated by the need to avoid the concurrent initiation of the stability mechanism by separate entities.

Strom and Yeminni [24] present a mechanism which supports the optimistic recovery of distributed computations. In their design, each node maintains a log of input messages which is periodically asynchronously committed to stable storage. A method of tracking causal dependencies is presented in which each process is aware of its causal dependencies and can

control appropriate recovery after failure. Recovery consists of finding some earlier checkpoint and replaying messages where possible. An implementation of these scheme based on the Mach operating system is described in [8].

Johnson and Zwaenepoel [11] provide an extended treatment which uses checkpoints and message logs to find a maximal recoverable state applicable to both optimistic and pessimistic logging protocols. The algorithms presented by Johnson and Zwaenepoel form the basis for recovery control in Grasshopper.

4. Grasshopper

Grasshopper needs to provide an intrinsic mechanism by which the system can always recover a consistent cut. In line with the goals of orthogonal persistence [1], this must be achieved without imposing the requirement that user code participate in the mechanism. Users of containers should only perceive a resilient persistent address space. Furthermore, Grasshopper seeks to provide this mechanism is such a way that individual container managers may use their own optimised stability protocols and still co-exist within the Grasshopper framework.

We have described how vector time can be used to represent the causal dependence between system components and further how it can be used to allow consistent cuts to be found. Extending Grasshopper to use vector time to provide the needed functionality occurs quite naturally.

To simplify discussion we will restrict ourselves to three entity types, namely loci, containers, and kernel. The remainder of this section describes the implementation of consistency in Grasshopper. We begin by considering a single node.

4.1. Vector time in Grasshopper

Conceptually the Grasshopper kernel maintains a vector for every container and every locus in the system. This vector contains as many entries as there are entities within the system. In reality, within the kernel vectors are represented by (entity id, time stamp) pairs. Initially, each entity's vector contains a pair representing itself. Additional pairs are added whenever entities interact.

The kernel implementation of time stamps is hidden from user level programs for two reasons:

> Entity names are an implementation dependant construct, there is no reason for their representation to be visible outside of the kernel.
>
> If user level code (such as in managers) has direct control over time stamps, malicious code could alter or forge them. This could result in user level code forcing the rollback of parts of the system over which they do not have authority.

The kernel therefore provides access to time stamps via capabilities which act as a proxy for the time stamp.

An optimisation may be made by observing that whilst executing, a locus is so intimately bound with its host container, that it makes no sense to distinguish between the time stamp of the locus and that of the host container. Therefore whilst executing, a separate vector for a locus need not be maintained. When the locus is not executing it ceases to be causally tied to the container. However, the maintenance of vectors during scheduling and de-scheduling of loci is very expensive. We therefore consider the bond between locus and host container to continue even when the locus is blocked. Once this step is taken there is no need to maintain a time stamp for a locus. The only proviso is that a locus must always have a host container, and therefore invocation must be an atomic operation. This atomicity is similar to that used by Clouds [4]. Since invocation is the only communication mechanism provided by Grasshopper, once invocation becomes atomic, it no longer makes sense for a consistent cut to cross a message send. This further simplifies the notion of a consistent cut.

A container's vector time is updated whenever it is invoked by a locus. When a locus leaves a container it takes with it a copy of that container's vector time. The kernel updates the invoked container's vector clock with this time using the normal vector time update mechanisms described above.

Mapping also causes containers to become causally interdependent. The act of mapping itself does not cause any change to either container's vector time. This occurs when a mapped region is accessed by an executing locus. A range of techniques may be used to update the

vector clocks when this occurs; these vary from simplistic coarse grain to complex fine grain mechanisms. These techniques are contingent upon the memory management architecture [16] and are therefore beyond the scope of this discussion.

4.2. Stabilisation

The key to resilient stable storage in Grasshopper is the interaction between the kernel and the container managers. Managers are responsible for creating recoverable copies of the contents of a container. It is the kernel that is responsible for the co-ordination of these stable states, creating and maintaining a recoverable system state. In a distributed system the separate kernel instances co-operate to maintain this recoverable state.

To this end the kernel must maintain in its own resilient store enough information to allow each container manager to recover the internal state of the containers in its charge. This must be done in such a way so as to ensure that after a system failure a self consistent system state can be rebuilt. A recovered system state need not be the same as any state that actually existed before a system failure, but it must be one of the possible system states, as might be have seen by some observer of the system.

4.3. The Stabilisation Sequence

Stabilisation is always mediated by the kernel. This ensures that the kernel has enough information about the stabilised state to make use of it in the event of failure. It also allows the kernel to direct other container managers to stabilise should it be following an eager policy.

Stabilisation may be initiated in a number of ways. The manager of a container may elect of its own volition to create a stable copy of the container's state. A stabilisation request to a manager may also be generated by the kernel in response to external factors. For example, pressure on kernel space, disk space, or rules that require recoverable states to be created within a given time may force commitment of a container's state.

Regardless of the source of the impetus to commit a container's state, the same stabilisation sequence is followed, and is always mediated by the kernel. Briefly, this sequence is as follows:

1. The kernel must ensure that the contents of the container does not change whilst stabilisation is taking place.
2. The manager creates an identifiable stable copy of the container.
3. The kernel creates a stable reference to the stable copy.

Firstly, the kernel protects the container from change. This may be achieved by de-scheduling all loci that are executing in the container or using memory protection. Next, the kernel then creates a stable copy of the container's time stamp and creates a new capability referencing it.

The next stage in the process is the responsibility of the container manager. The kernel therefore requests it to create a recoverable state of a container; this request includes the capability referencing the current vector time stamp of the container.

The manager builds an appropriate copy of the information needed to recover the current state of the container. The actual mechanism is not specified, and indeed much of the power of user level managers comes from this freedom. Mechanisms may include conventional shadow pagers, managers that perform compression, managers that do language specific transformation of the data such as pointer swizzling, encrypting managers and logging mangers. However the manager must always ensure that it is possible to recover all previous container states unless instructed otherwise by the kernel. At no time during the commitment of state, must the previous consistent container state be damaged. This ensures that even if a failure occurs during stabilisation, a usable stable state is always available.

The manager must include the capability reference of the vector time stamp in the committed state of the container since it is by this reference number that the kernel refers to committed states. Once the manager has created the stable state control returns to the kernel.

At this time, a complete stable checkpoint of the container has been established. Furthermore, the kernel has issued a vector time to be associated with that checkpoint. Using the capability for that time, the manager and the kernel can refer to a particular stable state. Therefore, over time, a sequence of checkpoints may be established for each of the containers in the system.

There is little point checkpointing the state of the system if the process executing within that system are not also saved. In Grasshopper saving the state of loci is the responsibility of the kernel. Whilst this may seem to be at odds with the policy of user level management of data, there is nothing to be gained in providing user level stabilisation of loci since the information to be saved is dictated by the underlying hardware. Stabilised loci are tagged using the same vector time as the container in which they are executing.

4.4. Stabilisation Policies

We have shown how a sequence of time-stamped checkpoints of system components may be established. To be useful on system recovery, a consistent cut must be formed from these checkpoints. The consistent cuts may be established by the kernel in a variety of ways. These are known as kernel *stabilisation policies*. In the current Grasshopper design policies are implemented within the kernel. It is hoped in the future, to be able to provide a mechanism which raises policies to the user level like managers. Policies may be placed in a spectrum ranging from eager to lazy.

Using a lazy policy, the kernel makes no attempt to find a consistent cut when the system is running. Individual managers stabilise the state of the containers under their control at will. On restart, the kernel traverses its collection of checkpoint sequences and attempts to find a consistent cut perhaps using the algorithm of Johnson and Zwaenepoel. At least one consistent cut must exist – the one in place at the previous system bootstrap.

Consistent cuts may also be created eagerly by the kernel. As described earlier such a policy may be used to ensure rapid recovery after a failure. Eager policies also ensure that the committed system state always reflects a recent state of execution. The most extreme form of eagerness is one where the external perception of the system is one of complete reliability, where no interactions with the system are lost due to failure. This requires some state to be committed on every external interaction.

Whenever a new consistent cut is established by the kernel, those managers that have contributed to it are notified. Notification includes the capabilities of all the obsolete stable states; that is all stable states older than the latest consistent cut. Managers use this information to reclaim the storage and internal data structures used to represent these states. The kernel is also free to reclaim storage. Firstly, it can discard the data associated with obsolete committed states. It is also able to compress the active time stamp vectors of all containers and loci which are part of the new consistent cut.

4.5. Distribution

As described earlier, the Grasshopper system seeks to provide an environment which abstracts over locality. For example, a locus may invoke, or map, any container for which it holds a capability; regardless of the node on which the locus is executing. This has the consequence that loci executing on many nodes may transparently and concurrently access the data of a container.

The user model presents the illusion of a single container that is available wherever it is required. In practice, this illusion is implemented by maintaining a representation of the container on each node. Earlier we stated that each container is maintained by a single manager. In reality, each local representation may refer to a separate manager. These managers collude to present the illusion of a single container and may be thought of as a single distributed manager. This coherent view is achieved by the managers exchanging pages with each other as necessary.

Each local representation of a container has associated with it its own time vector. Whenever a page is passed from one manager to another, a time vector is carried with it and the vector clock of the recipient local container representation is updated. Thus, causal dependencies arising through distributed shared memory require no further infrastructure.

A local container representation may be checkpointed independently of other representations. These checkpoints may only be used if the kernel(s) can find a consistent cut through the individual checkpoints upon recovery.

Alternatively the managers may cooperate to synchronously create a single coherent checkpoint; in doing so they will unavoidably produce a single vector time for all local representations.

4.5. Causal Delivery

The coherency of data stored in containers is not the only source of difficulty in a distributed system. Kernel mediated operations such as container mapping must also propagate to all other local container instances. Furthermore, the order in which these actions are perceived must be the same on each node

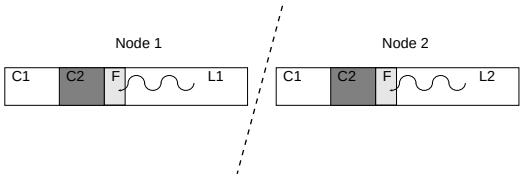

Figure 7: A Distributed Container

It is easy to construct examples where this is critical. Consider the example shown in Figure 7 in which Locus *L1* executing on Node 1 maps a region of container *C2* into container *C1*. After doing so, it updates a flag *F* in *C1* to indicate that the area has been mapped. *L2* is concurrently executing in *C1* on Node 2 and awaiting the completion of the mapping. Clearly the mapping must be established on Node 2 before the page containing the updated flag is inserted. In the general case, this requires causal ordering to be maintained on all globally visible operations.

This presents two problems; firstly, all local instances of a container must be identified. Secondly, a mechanism of causal delivery of the multicast operation must be provided.

The first problem is addressed by local instance managers. In order to present a single coherent view of a container they must have knowledge of each other's existence. The second problem may be addressed by the kernel making use of the vector time stamps to ensure causal delivery. The algorithm presented by Birman et al. [2] ensures causal delivery of multicast events even when propagated via intermediate nodes.

In Grasshopper, this is implemented by local instance managers identifying the recipients of system events and the kernels performing causal delivery.

5. A Few Final Observations

5.1. Self Managing Managers

Managers are user level programs, composed from containers and loci. Therefore mangers must have their own state managed by some manager and are intrinsically persistent. However, mangers may not use their own intrinsic stability to maintain stable state about containers in their charge. They must ensure that even if their own state were lost, enough information resides on recoverable media for a new instance of a manger to recover. The design of a manager may make use of the intrinsic advantages of the persistent programming paradigm for all other aspects of its operation.

The recursive composition of managers must be rooted in some fixed point. Eventually some manager must be found which is not dependant upon some other manager to support it. Grasshopper provides explicit support for a manger to manage its own data. This is achieved through a the provision of self-managing managers. For each self-managing manager, the kernel maintains a stable and resilient page of data. Upon system restart, this page will always be available in physical memory for the manager. This page provides sufficient data and code for the manager to bootstrap itself.

5.2. Deterministic Containers

Grasshopper provides a simple model of concurrent computation – multiple loci may execute and interact in a container. All synchronisation is assumed to be provided by language level constructs, which make use of system facilities. This results in a system in which execution within a container is intrinsically non-deterministic. However many of the above models of

providing stability depend upon the ability to rerun execution of programs deterministically. Grasshopper provides additional control over containers to allow these mechanisms to be used.

A container may be tagged as deterministic. Any such container is managed by the kernel in such a way that all execution is deterministic. This is achieved by ensuring the following two restrictions. Only one executing locus may exist within the container at once, and the container does not share any modifiable state (through mapping) with any other container.

The first restriction (single locus) leads to two possible strategies. Firstly, the kernel treats the container as a monitor which prevents more than one locus from executing within it. The second option is to allow many loci to execute within the container and guarantee determinism in the in the scheduler. This may be achieved without restricting the execution of all loci in the system by permitting loci which are not executing in deterministic containers to be scheduled using conventional algorithms. Thus deterministic replay can be guaranteed even when the apparent programming paradigm is concurrent. Care must be taken to ensure that one locus does not starve other loci.

The second restriction (no sharing of modified data) may also be provided in one of two ways. Firstly, the kernel can reject mapping requests which would allow modifiable data to be shared with a deterministic container.

Alternatively, if the contents of a deterministic container are committed to stable store before any modified state becomes externally visible, deterministic replay is still possible. This is similar to a mechanism proposed by Wu and Fuchs for providing hardware support for persistence [27]. Although apparently expensive this mechanism is no costlier than many transaction models proposed for persistent systems, and since only pages modified since the last such commit need be copied the expense need not be great.

6. Conclusions

In this paper we have described a new distributed operating system which is designed to support orthogonal persistence. Of particular importance in such a system is the consideration of causal interdependencies between components. We have shown how the paradigm of vector time may be used to characterise causal relationships in distributed systems in general and in the Grasshopper system in particular.

Central to the Grasshopper design are the concepts of containers, an abstraction over all memory and managers which support that abstraction. Above the manager abstraction all data appears to be stable and globally consistent. This illusion is provided by managers and the Grasshopper kernel(s) working in co-operation.

The Grasshopper system is designed to allow flexibility in the way in which persistence is provided. A key element of this flexibility is that individual managers may provide stability for the containers they manage in any way which they see fit. Managers are not concerned with issues of global consistency. This is the duty of the Grasshopper kernel(s) which are responsible for maintaining or finding globally consistent states using vector time. Further flexibility is provided by allowing the kernels to implement either eager or lazy consistency policies. We are currently searching for ways in which these policies may be implemented at the user level.

Acknowledgements

This train of research was inspired by discussions with Mike Livesey of St Andrews University during the one of the author's study leave. For those discussions we thank him.

References

1. 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.
2. Birman, K., Schiper, A. and Stephenson, P. “Fast Causal Multicast”, 1990.
3. Brown, A. L. “Persistent Object Stores”, Ph.D Thesis, 1988.

4. Dasgupta, P., LeBlanc, R. J. and Appelbe, W. F. “The Clouds Distributed Operating System”, *Proceedings, 8th International Conference on Distributed Computing Systems*, 1988.

5. Dearle, A., Bona, R. d., Farrow, J., Henskens, F., Hulse, D., Lindström, A., Norris, S., Rosenberg, J. and Vaughan, F. “Protection in the Grasshopper Operating System”, *International Workshop on Object-Orientation in Operating Systems*, to appear, 1993.

6. Fabry, R. S. “Capability-Based Addressing”, *CACM*, vol 17, 7, pp.403-412, 1974.

7. Fidge, C. “Timestamps in Message-Passing Systems That Preserve Partial Ordering”, *11th Australian Computer Science Conference*, University of Queensland, pp.56-66, 1988.

8. Goldberg, A., Gopal, A., li, K., Strom, R. and Bacon, D. “Transparent Recovery of Mach Applications”, *1st International Mach Workshop*, Vermont, 1990.

9. 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.

10. 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.

11. Johnson, D. and Zwaenepoel, W. “Recovery in Distributed Systems Using Optimistic Message Logging and Checkpointing”, *Journal of Algorithms*, vol 11, 3, pp.462-491, 1990.

12. 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.

13. Lamport, L. “Time, Clocks, and the Ordering of Events in a Distributed System”, *CACM*, vol 21, 7, pp.558-565, 1978.

14. Lampson, B. *Distributed Systems Architectures and Implementation*, vol 105, pp.250-265,

15. Li, K. “Shared Virtual Memory on Loosely Coupled Multiprocessors”, Ph.D. Thesis, 1986.

16. Lindström, A., Dearle, A., Rex diBona, Farrow, M., Henskens, F., John Rosenberg and Vaughan, F. “A Model For User-Level Memory Management in a Distributed, Persistent Environment”, *17th Australian Computer Science Conference*, Christchurch, New Zealand, pp.to appear, 1994.

17. Liskov, B. H. and Zilles, S. N. “Programming with Abstract Data Types”, *SIGPLAN Notices*, vol 9, 4, 1974.

18. 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.

19. Mattern, F. “Efficient Distributed Snapshots and Global Virtual Time Algorithms for Non-FIFO Systems”, 1990.

20. 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.

21. 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.

22. 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.

23. Schwarz, R. and Mattern, F. “Detecting Causal Relationships in Distributed Computations: In Search of The Holy Grail”, 1990.

24. Strom, R. and Yemini, S. “Optimistic Recovery in Distributed Systems”, *ACM Transactions on Computer Systems*, vol 3, 3, pp.204-226, 1985.

25. 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.

26. 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.

27. Wu, K.-L. and Fuchs, W. K. “Recoverable Distributed Shared Virtual Memory”, *IEEE Transactions on Computers*, vol 39, 4, pp.460 - 469, 1990.