Implementing the Comandos Virtual Machine

Vinny Cahill, Paul Taylor, Gradimir Starovic, Brendan Tangney,
Darragh O'Grady, Rodger Lea, Christian Jacquemot, Peter Strarup Jensen,
Paulo Amaral, Adam Mirowski, James Loveluck, Youcef Laribi,
Xavier Rousset de Pina and Pedro Sousa

Vinny Cahill, Paul Taylor, Gradimir Starovic, Brendan Tangney and Darragh O'Grady
Distributed Systems Group,
Department of Computer Science,
Trinity College Dublin,
Ireland
Email:Vinny.Cahill@dsg.cs.tcd.ie

Rodger Lea, Christian Jacquemot, Peter Strarup Jensen, Paulo Amaral and Adam Mirowski
Chorus Systèmes
6, Avenue Gustave Eiffel,
F-78182 Saint-Quentin-en-Yvelines CEDEX
France
Email:chris@chorus.fr

James Loveluck and Youcef Laribi
OSF Grenoble Research Institute
Gieres,
France
Email: loveluck@gr.osf.org

Xavier Rousset de Pina
Unite mixte BULL-IMAG,
Gieres,
France
Email: Xavier.Rousset@imag.fr

Pedro Sousa
INESC,
R. Alves Redol 9,
1000 Lisboa, Portugal
Email:pms@inesc.inesc.pt

Abstract

This report describes the different implementations of the the Comandos platform.

This report is published as Chapter 10. of *The Comandos Distributed Application Platform* Cahill, V., Balter, R., Harris, N. and Rousset de Pina, X. (Eds.), Springer-Verlag, Berlin, 1993.

Document No. TCD-CS-93-32

A strategic result of the Comandos project is the implementation and demonstration of a number of operational prototypes of the Comandos virtual machine, thus proving its feasibility in multi-vendor environments. Two basic approaches to the implementation of the platform were considered in the framework of the project:

- The first approach consisted of implementing the virtual machine as a guest layer on top of UNIX, without any modification to the UNIX kernel. One such implementation, Amadeus, was designated as the *reference platform* for the project. Therefore it was the basis for the integration of the numerous system components, application services and management tools developed throughout the project. This implementation is detailed in Sect. 1 below. The other major UNIX-based implementation of the virtual machine, IK, is described in [Sousa et al. 1993]. One of the objectives of the ESPRIT Harness project is to integrate this implementation of the Comandos platform with the DCE environment [HARNESS 1991b].
- The other approach followed was to implement the virtual machine directly on top of a micro-kernel. The motivation for this approach stemmed from the belief that the micro-kernel technology would be better able to support the Comandos abstractions, especially as far as distributed shared objects and protection were concerned. Note however that the target environments must, nevertheless, provide full access to UNIX applications. Two prototypes have been implemented using micro-kernel technology and are briefly described in Sect. 2 below:
  - one implementation runs on top of the CHORUS micro-kernel,
  - one implementation runs on top of OSF/1-MK.

These prototypes implement the VMI so that it will be possible to port a given language-specific run-time or service from one prototype to another. Full interworking of the various prototypes was beyond the scope of the project.

1 The Amadeus Platform

The Amadeus platform is the reference implementation of the Comandos virtual machine. Following the Comandos architecture, Amadeus consists of two main components: the GRT and the kernel. The Amadeus kernel is implemented as a collection of servers at each node, while the GRT is implemented as a library which must be present in every context. As a general rule, functionality is placed in the GRT whenever possible so as to maximise performance by avoiding inter-process communication and extra context switching. Functionality is assigned to the servers only if it must be protected. In effect, the servers provide the minimum functionality that would be implemented in protected mode if Amadeus were implemented on bare hardware or using an underlying micro-kernel.

The Amadeus kernel is a trusted component implemented by a number of servers. The so-called kernel server is responsible for cluster fault handling and extent management. On a cluster fault the kernel server is responsible for determining the extent, node and context in which the attempted invocation should be carried out and for forwarding the request to the target context as necessary. Thus the kernel server is responsible for maintaining the mapping between a cluster and its extent, for cluster location, and for extent activation and context creation. In addition to the kernel server, the RelaX Transaction Manager (TM), which is responsible for the execution of the distributed transaction coordination protocols, is implemented as a server at every node (c.f. Sect. 1.14). The TM and kernel servers make use of a totally ordered atomic broadcast protocol – the Reliable Broadcast Protocol

(RBP) – which is implemented by another server at each node. Finally, each node may support a collection of storage servers managing different types of containers.

The GRT is mainly responsible for generic object management, job and activity management and for cluster mapping and unmapping including interfacing to secondary storage. The GRT also has a role in cluster fault handling – enabling the kernel to be by-passed where allowable. Finally, the GRT is responsible for the synchronisation of accesses to and recovery of atomic objects,

Section 1.1 gives an overview of the main design decisions taken in Amadeus. Section 1.2 describes the internal components of the Amadeus kernel and GRT. Subsequent sections describe the design and implementation of these components of Amadeus. In the space available only an overview of the design can be given – the interested reader is referred to [Cahill et al. 1992] for a more detailed description of the design.

1.1 Basic Design Decisions and Assumptions

This section outlines some of the global design decisions taken in the implementation of the system and in particular explains how concepts such as user, context and container are mapped onto the facilities provided by typical UNIX systems.

Amadeus has been designed to run in a tightly integrated network of UNIX workstations connected by a local area network.

Each Amadeus user maps onto a specific UNIX user. It is assumed that each user has the same UNIX user identifier on all nodes in the network[1] and the same user identifier is used for the Amadeus user as for the corresponding UNIX user.

Although ideally an extent is defined in terms of a set of classes, in practice, in this UNIX implementation, each extent is defined by a text image which includes all of the classes which are permitted in the extent. A text image is associated with each extent when the extent is created (c.f. Sect. 1.5). In principle, a user can request different applications to be run in any extent. In practice, without dynamic linking, a user can only ask to run one application in any extent – that corresponding to the text image for that extent.

Each Amadeus context is implemented as a UNIX process known as a *client*. A client can be created either as a result of a user starting an application or because of a cluster fault by an existing application. The UNIX text image to be run by the client is determined from the extent represented by the client. When an application is started in the extent, the mainline of the image is run. However, when a client is forked as a result of a cluster fault, the client idles waiting for the cross-context invocation request rather than executing the application mainline. Note that there may be many client images available in any Amadeus system and many jobs executing simultaneously and running the same or different images in the same or different extents.

The kernel servers are also implemented as UNIX processes. Moreover, in order to be able to implement the necessary security mechanisms, at least the kernel server must run with UNIX `root` privileges e.g. to make use of the UNIX `setuid` system call.

Although activities may share objects there is no physical sharing of memory in the current implementation. Each cluster appears in only one context and shared objects are accessed only by cross-context invocation. The

[1] For example, because the UNIX network has a single password file shared by all nodes.

decision not to share objects between contexts was motivated by the desire to support the use of virtual addresses as object references in language level objects. An alternative to the use of multiple contexts might have been the use of a single context per node. However this was rejected for security reasons, because of the limitations on the size of virtual address spaces that can be sufficiently supported by standard UNIX systems and to increase failure isolation when applications crash.

Activities are represented by threads in each visited context.

The current version of Amadeus assumes an underlying distributed file system which is shared by all the nodes in the Amadeus system and is used to make control information which is required by the various kernel components globally available. Although the SS allows different implementations of containers, in the current version of Amadeus each container is implemented as a directory in the underlying file system, with each cluster in the container stored as a file in the corresponding directory.

Finally, the design assumes that an authentication service exists allowing the receiver of any message to authenticate the sender. It is expected that an authentication service suitable for the needs of Amadeus can be implemented using, for example, the Kerberos authentication service [Steiner et al. 1988].

1.2 The Structure of Amadeus

This section presents an overview of the internal architecture of the kernel and the GRT and the functionality of their various sub-components.

1.2.1 Kernel Structure.

The logical structure of the GRT and of the kernel have already been described in Chap. 9. of [Cahill et al. 1993] . The roles of the modules making up the kernel are introduced here and described in more detail in later sections. Figure 1 shows the main components of the kernel and their sub-components.

The kernel part of the PS is itself composed of a number of distinct modules. The kernel Extent Manager (EM) is responsible for all aspects of extent management including maintaining a database of known extents and their attributes, and also maintaining the mapping from a cluster to the extent to which it belongs. The EM also manages extent activation. The kernel Auditor is responsible for audit data generation for kernel operations.

The kernel part of the VOM is composed of four modules. The kernel Cluster Fault Handler (CFH) is responsible for resolving cluster faults including arranging for clusters to be mapped when necessary and forwarding of invocation requests to the appropriate contexts. The Location Service (LS) is responsible for keeping track of mapped clusters and ensuring that each cluster is mapped into at most one context at any time. The Address Resolution Manager (ARM) is responsible for keeping track of contexts and returning the address of a specified context when necessary. The kernel Context Manager (CM) manages context creation and deletion.

The kernel ES consists of the kernel Load Balancer (LB) which may be used to choose the node at which a faulted cluster is mapped, and the Thread Manager which provides support for concurrency within the kernel.

The kernel SS includes the Storage Manager (SM) which keeps track of which containers are available in the system at any time and also provides the interface between the kernel server and individual storage servers.

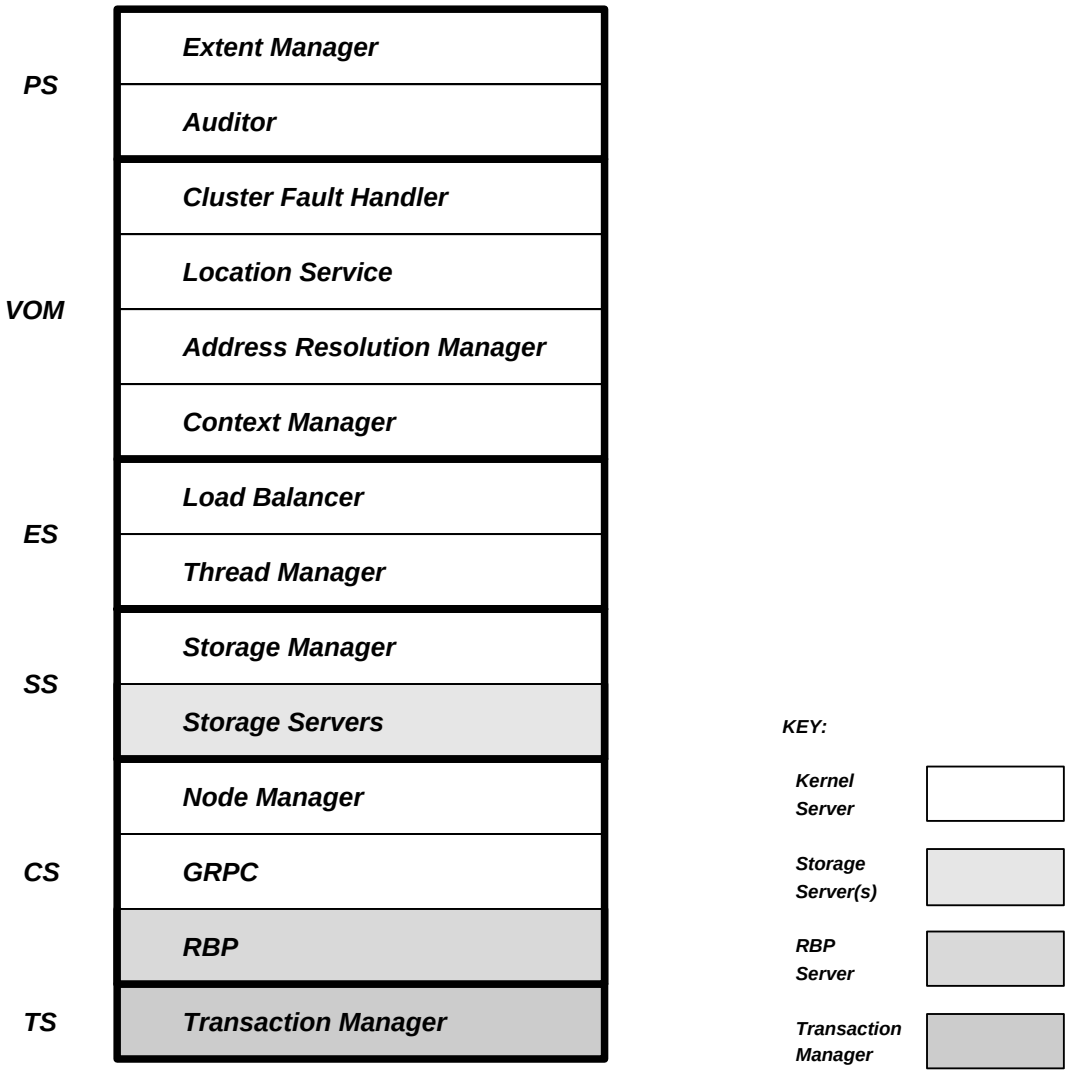

Figure 1: The components of the Amadeus kernel.

The kernel CS provides secure local and remote communication including reliable broadcasting and also includes the Node Manager (NM) which is responsible for keeping track of the nodes in the system.

Finally, the kernel TS supports distributed transaction management.

1.2.2 GRT Structure.

The main components of the GRT are outlined in Fig. 2 and discussed in detail in later sections of this report .

The GRT PS includes only the Auditor which is responsible for audit data generation for GRT operations.

As in the kernel, the GRT part of the VOM is made up of a number of different modules. The Object Manager (OM) is responsible for all operations related to the management of objects. The Cluster Manager (ClM) is responsible for cluster creation, mapping and unmapping. On a cluster fault the GRT CFH is responsible for initiating the mapping of the target cluster or the forwarding of the invocation request (possibly via the kernel CFH) to the target cluster. The GRT CM is responsible for the initialisation and termination of the context.

.

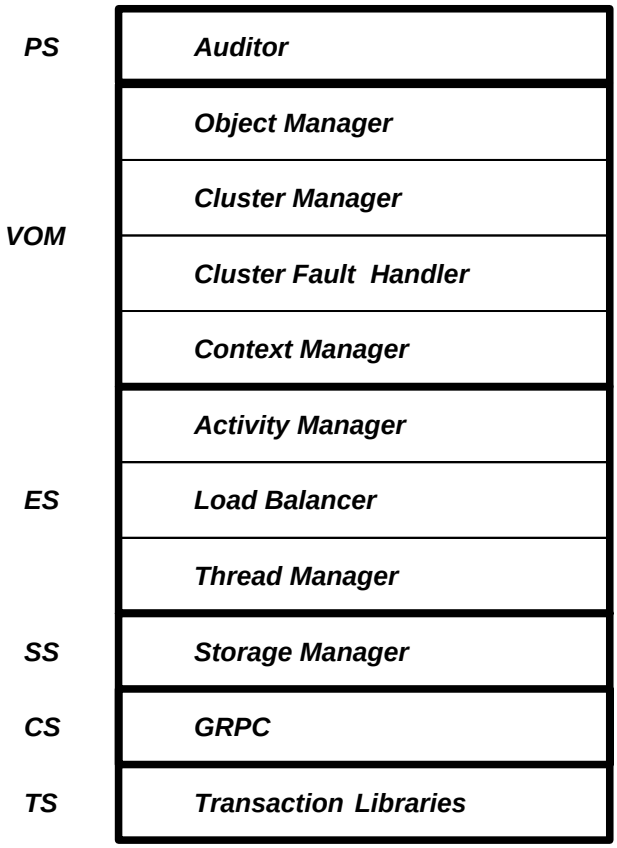

Figure 2: The components of the Amadeus GRT.

The GRT ES includes the Activity Manager (AM) which implements jobs and activities and the operations to control these. The AM also implements the cross-context invocation service and some low-level synchronisation

facilities. The ES also includes the GRT LB, which may be used to choose the node at which the clusters used by an activity are to be mapped, and the GRT Thread Manager.

The GRT SS provides the interface between the GRT and the storage servers.

The GRT TS is responsible for the synchronisation of accesses to, and recovery of, atomic objects. In particular, the TS encapsulates the RelaX transaction management libraries.

Finally, the GRT CS supports secure local and remote communication.

1.3 Node Management

An Amadeus *system* consists of a dynamically varying collection of nodes each potentially having a different architecture and running a different version of UNIX. Each node involved in an Amadeus system must run the kernel server. A node *joins* the system when the kernel server is first started on that node. A node *leaves* the system when its kernel server terminates. In general, a node may join or leave the system at any time.

Each node in an Amadeus system is identified by its *node name* which must be unique within that single system. Normally names are assigned statically to nodes by the local (Amadeus) system administrator and the name of the node passed as a command line argument to the kernel when it is started.

Each node also has a network address which is the address used by remote nodes to communicate with the kernel on that node. The form of the network address is known only within the CS and depends on the protocol stack in use in the system.

1.3.1 System State and the Active Node Table.

Each node maintains a certain amount of information describing the global state of the Amadeus system. The *system state information* describes both the nodes that are currently part of the system; the containers that are currently available in the system (c.f. Sect. 1.4.2) and the nodes that are responsible for controlling the mapping of clusters from each container (c.f. Sect. 1.7.1). In particular, the system state information includes the Active Node Table (ANT), described below; the Mount Table, described in Sect. 1.4.2 and the Control Node Table, described in Sect. 1.7.2. Note that the system state information is fully replicated at each node.

In order to keep track of which other nodes are part of the system and also to maintain information about certain characteristics of those nodes the NMs on each node maintains a local replica of the ANT. The ANT contains one entry for each node that is currently part of the system. Each ANT entry contains:

- the name of the node;
- the current ranking of the node in the system;
- the network address of the node;
- the UNIX (string) name of the node;
- the internet address of the node;

- the architecture of the node, and
- the current load on the node.

The ANT is maintained as a simple one dimensional array indexed by node name.

1.3.2 Node Manager Operation.

The kernel NM design relies on the use of totally ordered atomic broadcast and node failure notification mechanisms provided by the RBP. Nodes are ranked according to the order in which they were seen to join the system.

When a kernel is started, its NM takes control to initialise each of the components of the kernel using a configuration file for the node. In particular, this initialisation may include the mounting of some containers at the node (c.f. Sect. 1.4.2).

The NM must then advertise the presence of the node and its initial state to the other nodes in the system. In addition, the NM must obtain an up-to-date copy of the system state information.

The joining NM broadcasts a message to advertise its presence to the other nodes, allowing them to update their copies of the system state information appropriately.

The NM on the highest ranked node (if any) is responsible for providing the up-to-date system state to the joining node by broadcasting a reply containing its updated ANT, Mount Table and Control Node Table. Should a failure of the highest ranked node occur after the original message is received but before the reply is sent, the next highest ranked node (if any) takes responsibility for transferring the system state to the joining node.

When a node is about to leave the system the NM broadcasts a message to the other nodes allowing them to update their copy of the system state information including adjusting the rankings of each node.

In addition each node periodically broadcasts a message containing its current values of all the information, such as the load on that node, that is subject to periodic update.

1.3.3 Node Management Interface.

The NM provides an interface to the other kernel components which allows them to obtain a list of the nodes which are currently active and to obtain information about each active node such as the type of the node and the current load on the node. It is also possible to query whether or not a particular node is currently active. A subset of this interface is also available to applications via the GRT.

1.4 The Storage Sub-system

The SS provides facilities for the storage and retrieval of clusters, and for the allocation of system-wide unique identifiers for clusters and objects.

The fundamental requirement that had to be satisfied in the design of the SS was to allow different implementations of containers, and hence cluster storage, to coexist in a single Amadeus system. Implementations making use of the underlying file system, or bypassing the file system and using raw disk, or based on an alternative store such as an existing DBMS are envisaged. In particular, a container can be implemented as:

- a UNIX directory. Each cluster is stored as a file in that directory.
- a raw disk partition. In this case the SS can determine where on disk each cluster is stored. Moreover each cluster can be stored contiguously on disk.
- a large file. As above, except that the container is a single, large file instead of a disk partition; the underlying file system determines the physical location of the file on disk.
- a front end to a storage system such as a DBMS.

The current SS implementation uses the underlying UNIX file system to implement containers and clusters. Another implementation, based on raw disk and supporting replication, which will allow physical cluster and object identifiers[2] to be supported is currently being implemented. This implementation is targeted at applications which, for efficiency, require direct access to objects on disk.

1.4.1 Architecture.

The SS at any node consists of two major components – the *Storage Manager* (SM) and a collection of *storage servers*. The SM provides the interface between the SS and its clients. Storage servers implement containers.

The main function of the SM is to hide the implementation of containers, by directing requests to the appropriate, possibly remote, storage server.

Storage servers will be provided for each type (i.e. implementation) of container supported by the SS. At a given node a single storage server handles all requests for local containers of one type.

The SM has components in both the Amadeus kernel and in each GRT at a given node. The functionality of the kernel SM is a superset of that of the GRT SM since the kernel is responsible for initialising the local SS and maintaining data structures such as the node's mount tables (c.f. Fig. 3)[3].

A given storage server is local to a single node. However, requests to a storage server may come from both local and remote SMs. It is assumed that all requests received by a storage server can be authenticated.

Each storage server is implemented by a UNIX process. Communication between a SM and a storage server uses the Amadeus CS (c.f. Sect. 1.15). Each storage server has a unique network address which is used by both the local SM and all remote SMs to communicate with that server. A storage server is started by the local Amadeus server SM when the first container of the corresponding type is mounted at the node. Each supported container type is identified by a code. The mapping from this code to the text image to use for the corresponding storage server is maintained in a system configuration file.

[2] Identifiers which give a hint as to the storage location of the cluster or object on disk.

[3] The main clients of the SS are the CFH and NM in the kernel and the ClM in the GRT.

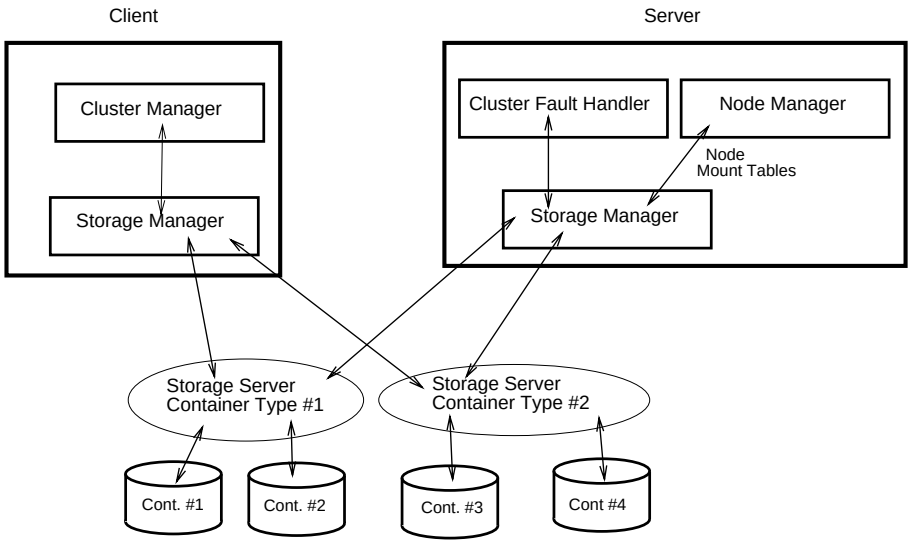

Figure 3: The SS architecture.

The Generic Storage Server. The primary function of a storage server is to provide the SM with a suitably high level abstraction of a container. This enables complex container implementations to be used without requiring that the SM be modified whenever a new container implementation is to be added. Each storage server must provide the following services:

- Cluster retrieval and storage;
- Cluster size query;
- Allocation of new cluster and object identifiers;
- Creation of new containers of the appropriate type;
- Mounting and unmounting of containers.

Individual storage servers maintain their own internal state including a list of the containers which they currently have mounted and the information needed to access the disk storage corresponding to each such container.

The Default Storage Server. Although the architecture of the SS allows specialised storage servers to be used, it is expected that a simple implementation based on direct use of the underlying (distributed) UNIX file system is sufficient for the needs of most applications. Moreover, given that the underlying file system provides an authorisation mechanism, it is possible to integrate such a storage server directly into the kernel and the GRT thereby avoiding the overhead of using the CS for communication between the SM and the storage server. The default storage server, which is always present on every node, is implemented in this way.

This implementation can be viewed abstractly as having the storage server existing within the SM (c.f. Fig. 4). In practice, this means that if a container implemented using the file system – a *UFS container* – is stored at the node on which the GRT is running, the GRT can access that container directly by making a system call (c.f. Fig. 6). If the container is not stored at the node, then the GRT must communicate directly with the remote storage server for the container i.e. the kernel at the node where the container is located (c.f. Fig. 7).

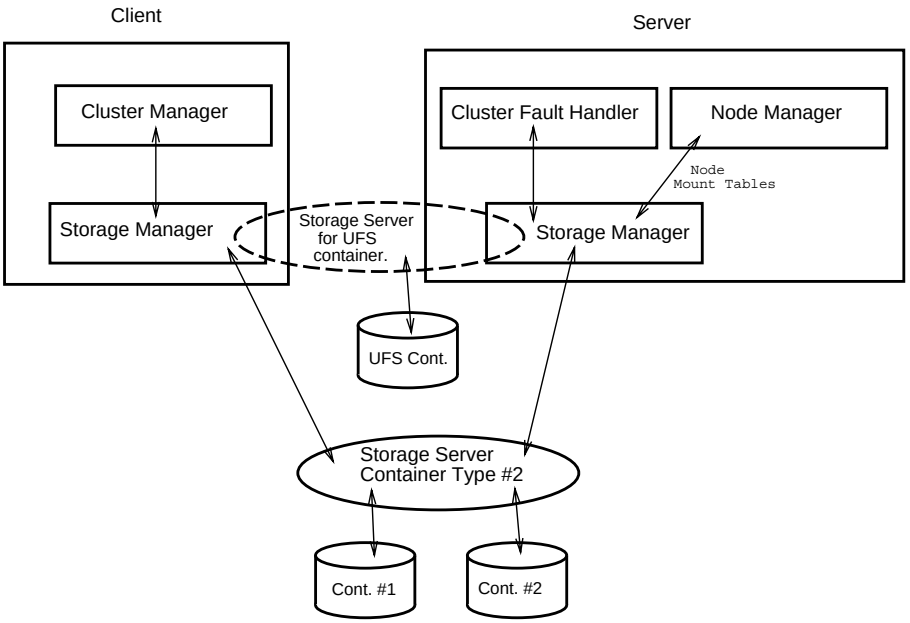

Figure 4: The default storage server.

1.4.2 The Storage Manager and Container Management.

The following sections describe the management of containers by the SM.

Supporting Multiple Storage Nodes for a Container. Normally, a container is stored at one node. A replicated container must, of course, be stored at several nodes. Moreover, a container that is stored using a distributed file system, such as AFS or NFS, may appear to be stored at several nodes i.e. at those nodes that have imported the volume storing the container[4]. In order to cater for each of these cases Amadeus admits the possibility that a container may be mounted at several nodes – its *storage nodes* – simultaneously. Such a container is managed by a different storage server at each node at which it is mounted.

In the case of a replicated container, each storage server maintains an independent copy of the container. It is the responsibility of each storage server to ensure that its copy of the container is consistent with those of other storage servers.

In the case of a container stored in a distributed file system, only one copy of the container exists, but the underlying file system gives the impression that a copy of the container exists at every node where the file system volume is mounted. Each storage server can treat the container as being stored locally.

Whether or not a particular container can be mounted at several nodes depends on its type. If a given container is of an appropriate type the SM will allow that container to be mounted by more than one node at a time. The system configuration file indicates which container types support multiple mounting.

[4]It may actually be stored at a single node or on several nodes – if the distributed file system supports volume replication.

```
typedef struct {
  ContainerName      lc;           /* container identifier */
  ContainerType      lctype;       /* type of container */
  ContextAddressType server;       /* address of server */
  char               directory[SS_UFS_MaxPath];
} LMTEntry;

typedef struct {
  ContainerName      lc;        /* container identifier */
  ContainerType      lctype;    /* type of container */
  NodeNameType       node;      /* node at which lc is mounted */
  ContextAddressType server;    /* address of server */
} MTEntry;
```

Figure 5: Mount Table Data Structures.

Data Structures. Whenever the SM receives a request for some container it must determine which storage server is to handle that request. The storage server may be either local, if the node is a storage node for the container, or remote. In any case, the SM at each node must maintain a mapping from a container name to the address(es) of the storage server(s) responsible for that container.

The kernel SM at each node maintains a table – the Local Mount Table (LMT) – which stores, for each container that is currently mounted at the node, the code identifying the type of container and the address of the local storage server which handles requests for that container (c.f. Fig. 5).

If the container is implemented by the default storage server the address is empty. In this case the LMT is also used to store the local path-name of the directory actually being used to store the container – this information is used by the default storage server in the local kernel and GRT.

The LMT is shared by the SM components in the kernel and GRT at the node. The LMT is mapped into a shared memory segment created by the kernel SM and attached by each GRT's SM. Only the kernel SM is capable of updating the LMT; all other copies are read-only.

The SM at each node also maintains another table – the Mount Table (MT) – which is similar to the LMT, except that it contains an entry for every container mounted in the system. The MT gives the name(s) of the node(s) at which each container is mounted and the address(es) of the corresponding storage server(s). As in the case of the LMT, the MT is available to be read by the local GRT. The MT is part of the system state information (c.f. Sect. 1.3.1) and, as such, is replicated at each node.

Containers are usually mounted at a node at boot time and unmounted when the node leaves the system. Dynamic mounting and unmounting of containers is also supported. It is also possible for a new container to be created (and hence mounted) dynamically. In each case the LMT and MT must be updated and the change to the MT broadcast (atomically) to the other nodes in the system.

SM Initialisation. During node initialisation, the kernel SM is responsible for determining which containers are to be mounted locally and for starting the necessary storage server processes.

Once started, a storage server blocks until it receives a request to mount a container. A mount request gives the storage server all the information it requires to initialise the specified container.

The SM obtains the list of containers to mount from its local mount file and sends requests to the appropriate storage servers to mount each container. The mount file gives the name and type of each container to be mounted as well as container-type specific information which is passed to the storage server to locate the disk storage for the container.

On system shutdown the SM sends requests to unmount each mounted container to each storage server before requesting the server to shut itself down.

Mounting and Unmounting Containers. Containers are normally mounted on node startup, and are unmounted when the node shuts down. Containers mounted at a given node during startup are described in that node's mount file. Applications may also mount and unmount containers which are not described in the mount file. This will normally apply to movable containers such as floppy or optical disks.

There are four events which involve mounting or unmounting containers, as follows:

1. Node startup.
   All containers described in the node's mount file are mounted locally.

2. Node shutdown.
   All locally mounted containers are unmounted.

3. Container Mount.
   The specified container is mounted locally.

4. Container Unmount.
   The specified container is unmounted.

1.4.3 Naming.

Each container is known by a system-wide unique name. The SS is also responsible for the allocation of system-wide unique identifiers which are used to name clusters and objects.

An Amadeus unique identifier – `UId` – is guaranteed to be unique within a single Amadeus system. `UId`s are not reused. `UId`s are created by concatenating the name of a container with a generation number which is unique within that container. Note however that the exact structure of a `UId` is opaque to the other components of Amadeus, so the interface to the SS provides operations to compare `UId`s and to return the container name and generation number from a given `UId`.

The next available generation number for each container is stored in the corresponding container.

1.4.4 The (Distributed) UNIX File System Implementation.

In the current version of Amadeus each container is implemented as a directory, with each cluster that is stored in the container being implemented as a file in that directory. For each container the path-name of the corresponding directory is given in the LMT.

Given that the current implementation assumes an underlying distributed file system it is typical for each node participating in the Amadeus system to mount each container. In this case each container is mounted at every node so that all reads and writes are performed locally.

The `UId` for a cluster is mapped directly to the path-name for the file in which the cluster is stored. For example, a cluster with a `UId` having container name equal to 1 and generation number equal to 123 is stored in the file `/container_1_path/123`.

The next available generation number for each container is stored in a file in the corresponding container directory; this file is only accessible to root.

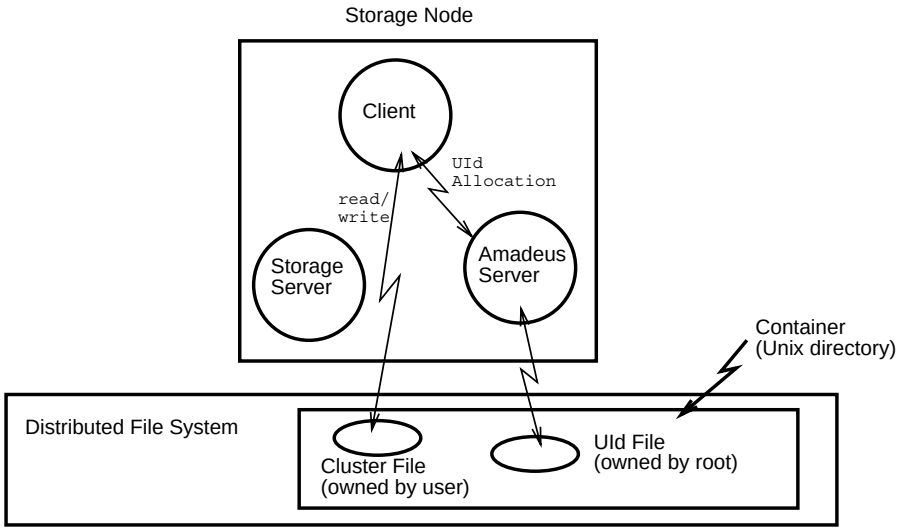

Figure 6: UNIX FS implementation: faulting node is a storage node.

Security. Since clusters are stored in a UNIX file system they are protected only by the usual UNIX mechanism of access control lists. Each cluster may be read/written by the client which is mapping/unmapping it. Hence each cluster should be both readable and writable by the user who owns the cluster (i.e. the user who owns the extent to which the cluster belongs) to allow mapping and unmapping of the cluster from a client running on behalf of that extent. The GRT storage server creates cluster files with read/write access for the cluster's owner (and, implicitly, the super-user). Even if the owner subsequently changes the protection assigned to the file, only the confidentiality and integrity of that user's own data can be effected if other users, who under Amadeus have no rights to the cluster, are allowed to access or alter the cluster outside of Amadeus. Note however that it is impossible to effect the confidentiality and integrity of data belonging to another user. Although resulting in weaker security, since the file containing the cluster must then be accessible outside of Amadeus, this approach allows the storage server to be integrated into the SM in the kernel and GRT.

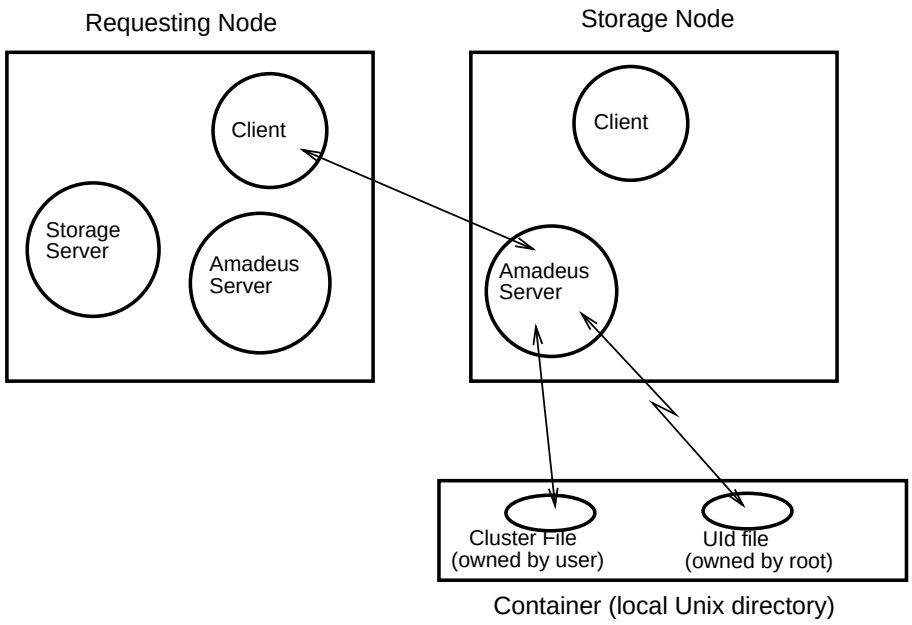

Figure 7: UNIX FS implementation: faulting node is not a storage node.

Those files storing control information, for example those which are used in the allocation of unique identifiers, must be protected against malicious damage. Hence these files are only accessible to root (i.e., the kernel) on each node. Thus only the storage server running in the kernel is able to allocate unique identifiers.

1.4.5 Storage Sub-system Interface.

The main interface to the GRT SM is that which is used by the ClM (c.f. Sect. 1.11) to read the named cluster into the current context at the specified address. The GRT SM also provides a corresponding routine to write a cluster that is being unmapped. Note that there is no primitive to create a new cluster. As described in Sect. 1.11.2, clusters are created in virtual memory and subsequently written to secondary storage.

The GRT SM interface also includes primitives to create, mount and unmount containers. However these routines are essentially wrappers for the corresponding kernel routines. The final parameter to the create and mount routines depends on the type of the container – for the file system implementation it is a path-name.

The kernel and GRT SMs also provide an interface which allows other kernel components to determine whether or not a given container is mounted and if so where. This interface is also available to applications via the GRT.

`UIds` are always allocated by a storage server. Moreover, allocation of `UIds` will be a frequent event. Hence, in order to reduce communication overhead, `UIds` are allocated by each storage server in blocks.

1.5 User and Extent Management

The PS at each node is responsible for all aspects of extent management including the creation of new extents and the activation of existing extents. The PS runs in protected mode although it offers an interface to applications via the GRT to, for example, create a new extent.

1.5.1 Amadeus Users.

Every registered user on any node that is taking part in an Amadeus system is a legitimate Amadeus user i.e. there is no need for a (UNIX) user to be registered with Amadeus before running an Amadeus application. Note, however, that each user must own an extent before running their first application.

1.5.2 Extent Naming.

Each user can have many extents. Each extent is known to its owner by an index number which is allocated by the PS when the extent is created. Extent index numbers are assigned chronologically to extents on a per-user basis.

Within the system each extent is known by a system wide unique name which is formed from the concatenation of the owner's UNIX user identifier and the index number of the extent with respect to its owner.

1.5.3 The Extent and Cluster Registers.

An Amadeus system must maintain information about each extent required for extent activation. At run time this information must be accessible to the PSs on each participating node but must otherwise be protected. In the current implementation, the information in the Extent Register is stored as a collection of files in a well-known directory in the underlying distributed file system with one file for each user – the name of the file being the user identifier.

For each extent the following information is maintained:

- the extent name (i.e. the identifier of the extent's owner and the index number of the extent with respect to its owner);
- a list of possible activation nodes for the extent (optional);
- the text image to be used for contexts of the extent, and
- the number of clusters that belong to the extent.

For cluster fault handling the PS also needs to be able to determine, given the name of a cluster, the name of the extent to which that cluster currently belongs. This information in the Cluster Register is also maintained as a collection of files in the underlying distributed file system with a single file holding the mapping information for all the clusters belonging to one container.

1.5.4 Extent Creation.

Currently, a request to create an extent can come from only one of two possible sources:

- the `mkextent` utility;

- an activity running in an existing extent.

Extent creation is handled entirely within the PS at each node since it requires access to the Extent Register. The operation to create an extent takes as parameters the identity of the user on whose behalf the extent is being created, a list of activation nodes and the text image to use for the extent. The name of the new extent is returned.

The only effect of creating a new extent is to make an entry in the Extent Register – initially a new extent contains no clusters and is not active anywhere.

`mkextent` is an interactive utility provided primarily to allow a new 'Amadeus user' to create his first extent. The `mkextent` utility can only be run at a node where the kernel is already running.

An extent created from an existing extent is owned by the owner of the creating extent. The remaining parameters are obtained from the parameters to the operation to create an extent.

1.5.5 Extent Activation.

An extent can have contexts at one or more nodes in the system.

A new context can be (but is not necessarily) created for an extent in one of two circumstances:

- as a result of a fault on a cluster belonging to the extent;
- when a user starts an application in the extent.

Invocation Requests. When the kernel CFH (c.f. Sect. 1.8) discovers that the target of an invocation request is not mapped, it calls the PS which is responsible for choosing a node at which the cluster can be mapped depending on the activation nodes (if any) specified for the cluster's extent.

The PS must determine the extent for the cluster and obtain the activation nodes for that extent. The PS simply chooses a node at which to map the cluster either from the list of activation nodes for the extent or by load balancing.

Given the node name returned by the PS the kernel CFH will forward the request to that node. When such a forwarded invocation request arrives at a node the local kernel CFH calls the local PS to determine the local context in which the cluster should be mapped and in which the request should be handled if any. The PS checks if a local context for the extent already exists. If not, a new context is created for the extent before the request is forwarded to the appropriate context.

Cluster Mapping Requests. In this case the kernel CFH calls the PS to determine if the target cluster can be mapped into the required context. This is possible only if the context represents the extent to which the cluster belongs.

Starting an Application. An application can be run from the shell by using the interactive utility `launch` which allows a user to specify both an application to run and the extent in which the application is to be started[5]. `launch` also collects any command line parameters to be passed to the application.

On receiving a request to start a new application, PS first verifies the requester's right to run the specified application in the specified extent[6]. If valid the request is treated in much the same way as an invocation request. The local PS is used to choose the node at which to start the application. The request is forwarded to the PS at that node which determines the context in which the request should be handled. If an appropriate context already exists then the request is simply forwarded to that context. If no appropriate context already exists, one will be created.

1.5.6 The Protection Sub-system Interface.

As well as creating a new extent, a user can also delete one of his own existing extents. However destruction of an extent is only permitted if there are no clusters in the extent.

A newly created cluster (c.f. Sect. 1.11.2) initially belongs to the extent in which it was created. However, clusters can also be moved between extents belonging to the same owner.

1.6 Context Management

An Amadeus system can be viewed as a collection of contexts dispersed throughout a distributed system, each containing a varying collection of clusters. Contexts can be created and deleted on demand as applications are started and invocations are made between contexts on the same or different nodes.

The main design goal was to isolate the higher levels of Amadeus from how contexts were implemented and, in particular, how contexts are addressed and communication between contexts performed.

1.6.1 Basics.

Contexts are created dynamically by each node as applications are started and cluster faults handled (c.f. Sect. 1.5.5).

Once created each context should persist at least until there are no active invocations in the context i.e. there are no activities present in the context. A context may persist longer if it is expected that further invocation requests may arrive in the 'near' future. It is also possible that a context may fail before all the activities that were present have completed, for example, as a result of a node failure.

Each context has a system wide unique name which is formed from the concatenation of the extent and node names. For communication between contexts to proceed each context must have a unique address. The format of the address depends on the inter-context communication mechanism in use and need only be known to the lowest levels of the CS (c.f. Sect. 1.15).

The remainder of this section discusses context creation, initialisation, and deletion in detail.

[5] Recall that currently the only application that can be run in an extent is that corresponding to the text image for the chosen extent.

[6] Currently only the owner of an extent can start an application in that extent.

1.6.2 Context Creation.

Contexts are normally created when a new application is started or as a result of a cluster fault. A failed context may also be restarted by the TS if the context contained the youngest committed version of any atomic object at the time of its failure and the object had not been written to secondary storage (c.f. Sect. 1.14).

The kernel CM provides a single primitive which can be use both to create a new context and to restart a failed context. Creating a new context basically involves `forking` a new UNIX process and having the new process `exec` the required code having first set the real and effective user identifiers of the new process to be those of the owner of the corresponding extent.

The new context is passed an indication of the circumstances in which it is being created to allow it to initialise itself correctly. In particular, if a new application is being started the mainline is run. If the context is restarting, then the TS takes control to recover the state of the context from its log as described in Sect. 1.14. Otherwise, the new context idles until an invocation request is received.

When a new context is created it must register itself with the kernel so that the kernel can begin forwarding requests to it. Note that a failure between the time a context is created by the kernel and the time that the context registers itself must be detected by the CM and propagated to the requester. Thus the CM, having created a new context, waits for that context to register itself (indicating successful initialisation) or for the new UNIX process to die (indicating unsuccessful initialisation). Once the context is registered, and depending on the flag passed as an argument to the process, the mainline may be executed as the initial invocation of a new job.

1.6.3 Context Deletion.

A context cannot be deleted while there are ongoing invocations present in the context. In order to keep track of when it is safe to delete a context, the GRT CM maintains a reference count for the context. When a context is created, its reference count is initialised to 0. Whenever a thread is created within the context to carry out an invocation either as the result of a job or activity creation, or the arrival of a cross-context invocation in the context, the reference count is incremented. The reference count is decremented as threads terminate. When the reference count reaches zero again, the context is not immediately deleted but rather a timer is set to delay context deletion for some period. If a new invocation request arrives in the context before the timer expires, the timer is cancelled and the request proceeds as normal having incremented the reference count.

When it has been decided to delete a context it is necessary that all mapped clusters be unmapped. Since the kernel cannot unilaterally unmap a cluster for the reasons explained in Sect. 1.10.6, it first up-calls the OM to inform it that the context has been terminated. The OM can then tidy up using the normal call to have each cluster unmapped by the CM in turn.

Finally, the kernel CM must be informed that the context is terminating.

1.7 Cluster Location

Central to the operation of Amadeus is that only a single image of any cluster may be mapped at any time i.e. a cluster is either mapped in exactly one context or is stored in the SS. The LS is responsible for keeping track of the current location of each mapped cluster. Moreover, the responsibility for ensuring that only a single image of

each cluster is mapped rests with the LS. The design of the LS must cater for the possibility that, due to a node crash, a mapped cluster may be *lost* from virtual memory or if not lost[7], may be unavailable while the node at which it was mapped is down.

1.7.1 Control Nodes.

Since it would be too expensive for each node to keep exact information about the location of every mapped cluster, information about the locations of mapped clusters is partitioned. A single node is assigned the responsibility of maintaining information about the current location of all the clusters from a given container. That node is known as the *control node* for the container. A given node may be the control node for zero, one or more containers. Although not strictly necessary, the control node for a container is always chosen from among the storage nodes for the container. In addition, every node keeps exact information about those clusters which are mapped locally.

The LS at a cluster's control node acts as the central authority in determining whether or not a cluster is mapped, thereby ensuring that a cluster can only be mapped once in virtual memory.

The control node for a cluster represents a single point of failure. If the control node for some container is down then it may not be possible to access clusters from the container even if they are mapped at another node [8]. It will certainly not be possible to map or unmap a cluster from the containers effected, even if it is stored at other nodes. Moreover, the information maintained by the LS must be resilient to node failures.

1.7.2 Data Structures.

The LS at each node maintains a table – the Control Node Table (CNT) – which gives the name of the control node for each container that is currently mounted. The CNT is part of the system state information (c.f. Sect. 1.3.1) and as such is fully replicated at each node.

Each control node maintains a table – the Mapped Cluster Table (MCT) – with one entry for each 'local' cluster that is mapped into virtual memory, giving the name of the node into which the cluster is mapped. The information, maintained in the MCT must survive node failures. Hence, the MCT is backed up on secondary storage and all changes to the MCT forced to disk. Note that, after any node failure, the MCT may not be correct since it may still contain entries for clusters that have been lost from virtual memory as a result of the failure. In this case, the consistency of the MCT is re-established, not when the failed node recovers, but lazily as attempts to access lost clusters are made.

Finally, each node maintains a table – the Kernel Cluster Table (KCT) – giving, for each cluster that is mapped on the node, the name of the (local) context into which the cluster is mapped. The KCT is lost on a node crash but rebuilt when the node recovers. On a context failure, the KCT is updated when the context is restored. The KCT is therefore always up to date. It is used both to speed the location of a cluster that is mapped locally and to validate the result of an MCT lookup which may be out of date.

[7] For example, because it will be recovered by the TS.

[8] Unless the requester has a valid MCC entry for the cluster.

1.7.3 Locating a Cluster.

The kernel CFH at the requesting node calls its local LS whenever a request for an absent cluster is received which may be either an invocation request or a request to map the cluster locally. The type of request is opaque to the LS whose role is to forward the request to the kernel CFH at the node where the target cluster is mapped, if any, or else, to the kernel CFH at the control node for the cluster.

When a cluster is required, the LS at the requesting node attempts to locate the cluster using only local information. If it cannot determine the location of the cluster, then the request is forwarded to the control node for the cluster (as given in the local copy of the CNT) which will have the necessary information to locate the cluster. The operation of the LS is summarised in Fig. 8 and Fig. 9 for the cases where the target cluster is not already mapped and is already mapped respectively.

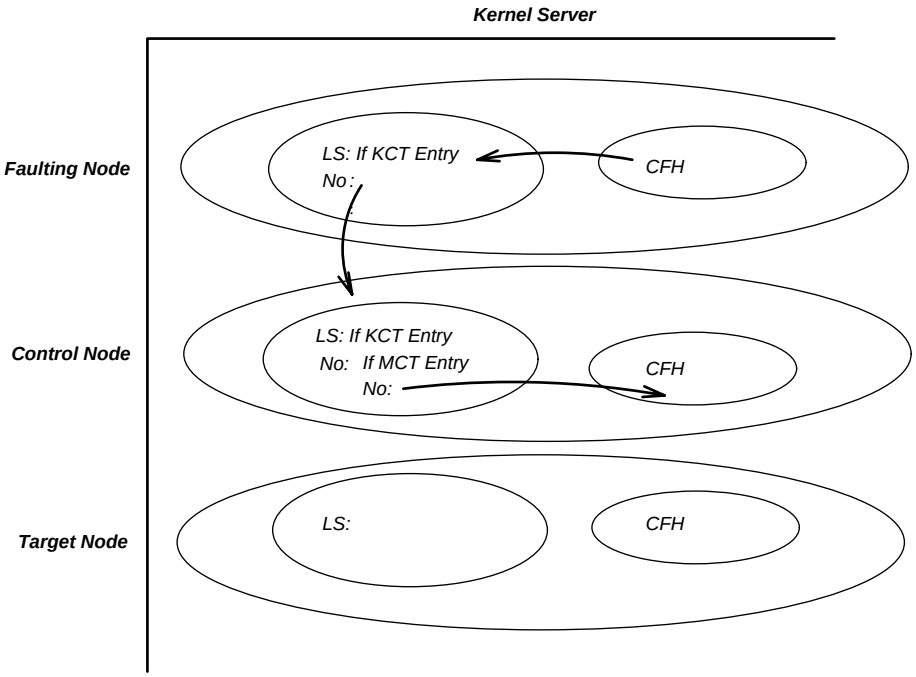

Figure 8: Operation of the LS when cluster is not mapped.

The Faulting Node. The LS first searches the KCT to determine if the cluster is mapped into a local context. If so the name of the context is returned immediately to the CFH.

If the cluster is not mapped locally but the node is the control node for the container, the local MCT is searched.

If an MCT entry for the cluster exists then the request is forwarded to the LS at the specified node.

If no MCT entry exists then the cluster is not mapped and the LS returns an appropriate indication to the local CFH.

Alternatively, if the node is not the control node for the required cluster, then the request is forwarded to the LS at the control node.

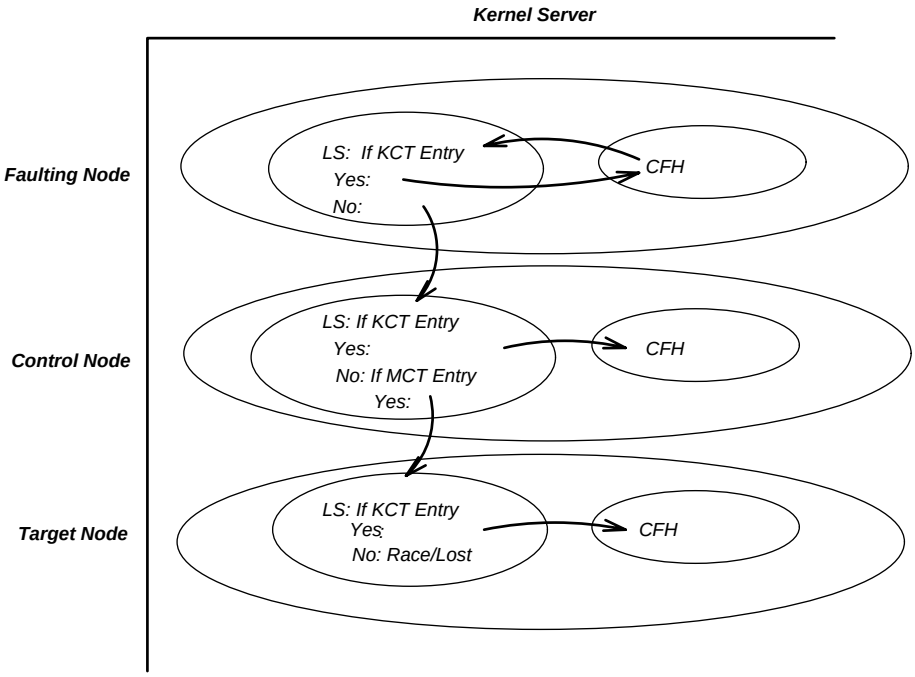

Figure 9: Operation of the LS when cluster is mapped.

If a request for a cluster is forwarded to a node (either the control node for the cluster or the node where the cluster is thought to be mapped) which is down, then the request will eventually timeout and an exception be raised in the calling activity.

The Control Node. If a forwarded request is received by the control node for a cluster, the sequence of events is similar to those that take place when the requesting node is also the control node.

The LS first searches the KCT to determine if the cluster is mapped into a local context. If so the local CFH is passed the request together with the name of the context.

If the cluster is not mapped locally the local MCT is searched. If an MCT entry for the cluster exists then the request is forwarded to the LS at the specified node. If no MCT entry exists then the cluster is not mapped and the LS passes the request to the local CFH.

The Mapping Node. When a request for a cluster is received from the control node for the cluster, by the LS at the node thought to be mapping the cluster, the KCT is searched for an entry for the cluster.

If an entry exists, the local CFH is passed the request together with the name of the context.

If no entry exists, then either, due to race conditions, the KCT entry for the cluster has not yet been made or the cluster has been unmapped since the request was forwarded from the control node, or, the cluster has been lost as a result of a previous node failure. In either case the request is returned to the control node which will then determine the status of the cluster.

1.8 Cluster Fault Handling

A cluster fault occurs when an attempt to access a cluster that is not mapped into the current context is made. Cluster fault detection is the responsibility of the OM when called to handle an object fault detected by the language-specific run-time.

1.8.1 The GRT Cluster Fault Handler.

On detecting a cluster fault the OM calls the GRT CFH. The OM may request that an attempted invocation be carried out at some node. The OM passes a description of the attempted invocation to the CFH in a standard form known as a `tblock`. The `tblock` header is interpreted by the CFH to obtain the name of the target cluster and other necessary information. The remainder of the `tblock` is opaque to the CFH but will be passed to the OM in the context in which the cluster is eventually mapped.

Alternatively the OM may request that the cluster be mapped into the current context – assuming that this is allowed by security and that the cluster is not already in use elsewhere.

Data Structures. The GRT CFH maintains a cache – the Mapped Cluster Cache (MCC) – which contains entries for clusters which are not mapped into the context but which have been recently accessed from this context. The MCC entry for a cluster identifies the context in which the cluster is thought to be mapped.

Handling a Cluster Fault in the GRT. For an invocation request, the `tblock` received is first tagged with the user and extent names, the preferred node for the requesting activity and the address of the context. The CFH then looks up the target cluster in its MCC.

If no MCC entry is found then the request is forwarded to the local kernel CFH. The kernel CFH will locate the context into which the cluster is mapped if any, or else, choose the context into which the cluster should be mapped. If a context other than the requesting context is chosen then the request will be forwarded to the GRT CFH in that context. The invocation will be dispatched to the target object via the OM, the operation carried out and the results returned directly to the requesting context using the context address in the `tblock`. In addition, the identity of the context in which the invocation was carried out is returned with the results and used to update the caller's MCC. The operation of the kernel CFH is discussed in more detail below.

If a MCC entry is found, the request is forwarded directly to the GRT CFH in the specified context where it is passed up to the local OM (c.f. Fig. 10).

If the cache entry was valid, the request will be carried out in that context and the results returned to the GRT CFH in the requesting context using the address given in the `tblock`.

If the cache entry was invalid, the OM in the target context will refuse the request. The local GRT CFH will then treat the request in the same way as an invocation request originating in that context and handle it in the same manner except that the `tblock` will contain the address of the original requesting context so that when the invocation is eventually carried out the results can be sent directly to the caller. In addition, the caller will then be able to update his cache with the address of the context where the invocation was carried out.

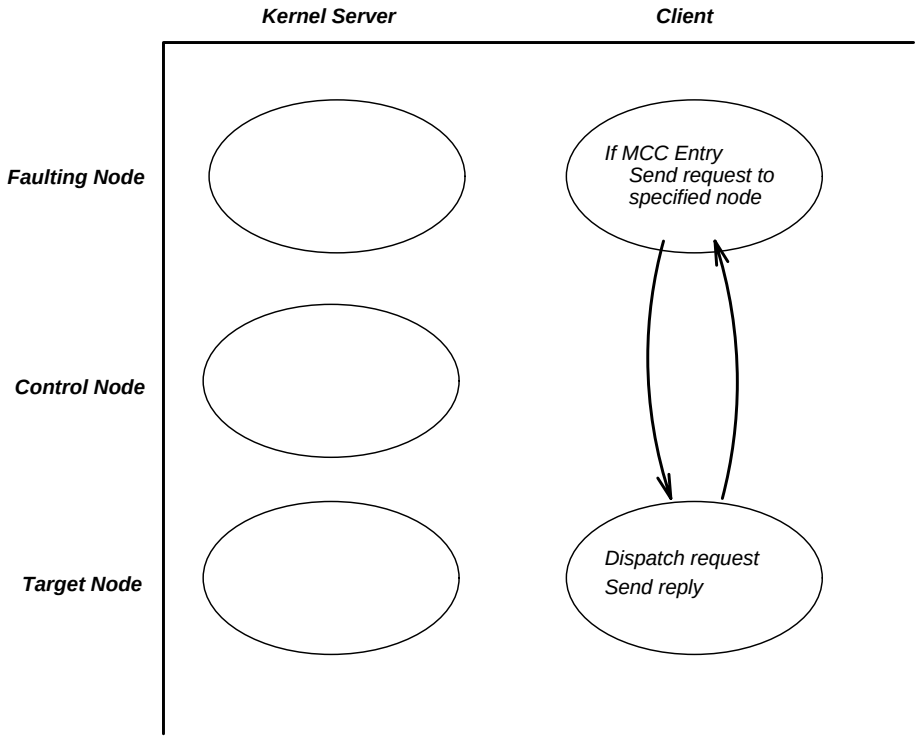

Figure 10: A cluster fault handled within the GRT.

When a request to map a cluster is received, it is passed immediately to the kernel CFH without looking for an entry for the cluster in the MCC.

The kernel CFH will locate the context into which the cluster is mapped if any, or else, arrange for the cluster to be mapped into the requesting context if permissible. If the cluster is already mapped the GRT CFH in the requesting context is informed. The operation of the kernel CFH is discussed in more detail below.

1.8.2 The Kernel Cluster Fault Handler.

The kernel CFH is called by the GRT CFH in one of two circumstances.

- When the GRT CFH receives an invocation request for a cluster for which it has no MCC entry.
- When the GRT CFH receives a request to map a cluster.

Handling a Cluster Fault in the Kernel. In either case the kernel CFH uses the LS to forward the request either to the kernel CFH at the node where the cluster is mapped or, if the cluster is not mapped, to the kernel CFH at the control node for the cluster (c.f. Sect. 1.7).

If the cluster is already mapped (c.f. Fig. 11), the kernel CFH at the mapping node simply forwards the request to the GRT CFH in the context specified by the LS.

If the cluster is not mapped (c.f. Fig. 12), the kernel CFH at the control node must arrange to have the cluster mapped in some context and the request forwarded to the chosen context.

At the control node, the PS is called to determine the node at which to map the cluster depending on the extent to which it belongs and the activation policy for that extent. The local LS is informed where the cluster is being mapped and the request forwarded to that node. If the target node is not a storage node for the cluster, the cluster can be read at the control node and appended to the request before it is forwarded to the kernel CFH at the chosen node.

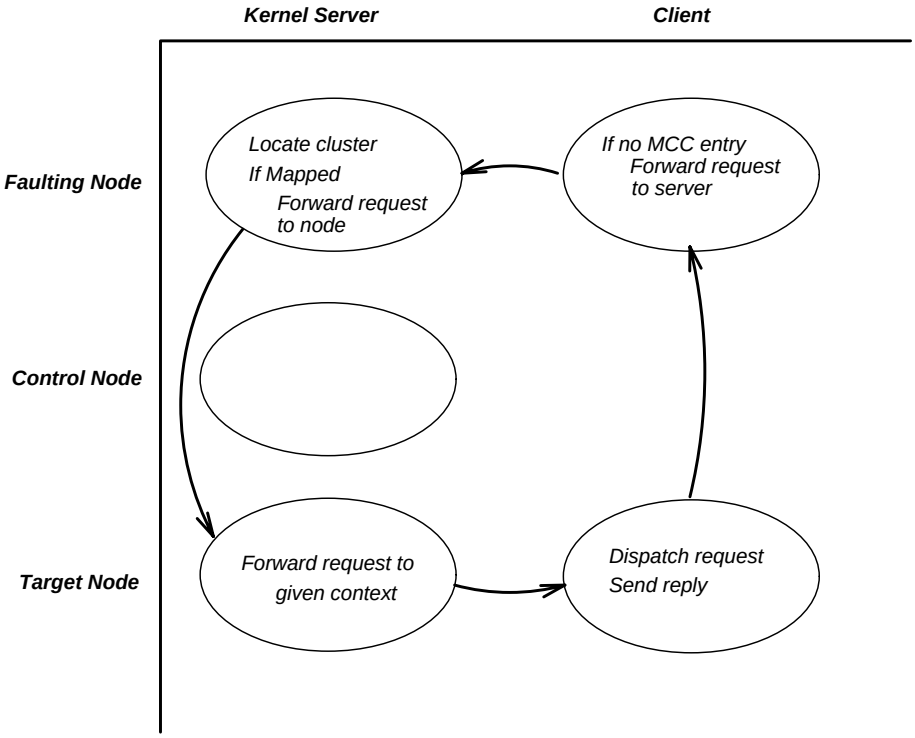

Figure 11: Cluster fault handling when cluster is already mapped.

At the chosen node the kernel CFH calls the PS to determine the context into which to map the cluster. The local LS is informed and the request forwarded to the GRT CFH in the chosen context. In this case the GRT CFH, must call the ClM to map the cluster before dispatching the invocation via the OM.

Once the operation has been carried out, the results are returned directly to the requesting context using the context address from the request. Moreover, the identity of the context in which the request was carried out is also returned to allow an MCC entry for the cluster to be made in the requesting context.

In the case of a request to map a cluster if the cluster is already mapped, the kernel CFH at the mapping node simply returns an error indication to the GRT CFH in the requesting context.

If the cluster is not mapped, the kernel CFH at the control node queries the PS as to whether or not it is permissible to map the cluster into the specified context. If so, the LS is informed where the cluster is being mapped and the kernel CFH at the requesting node is informed to proceed with mapping. Optionally, if the requesting node is not a storage node for the cluster the cluster can be read at the control node and returned to

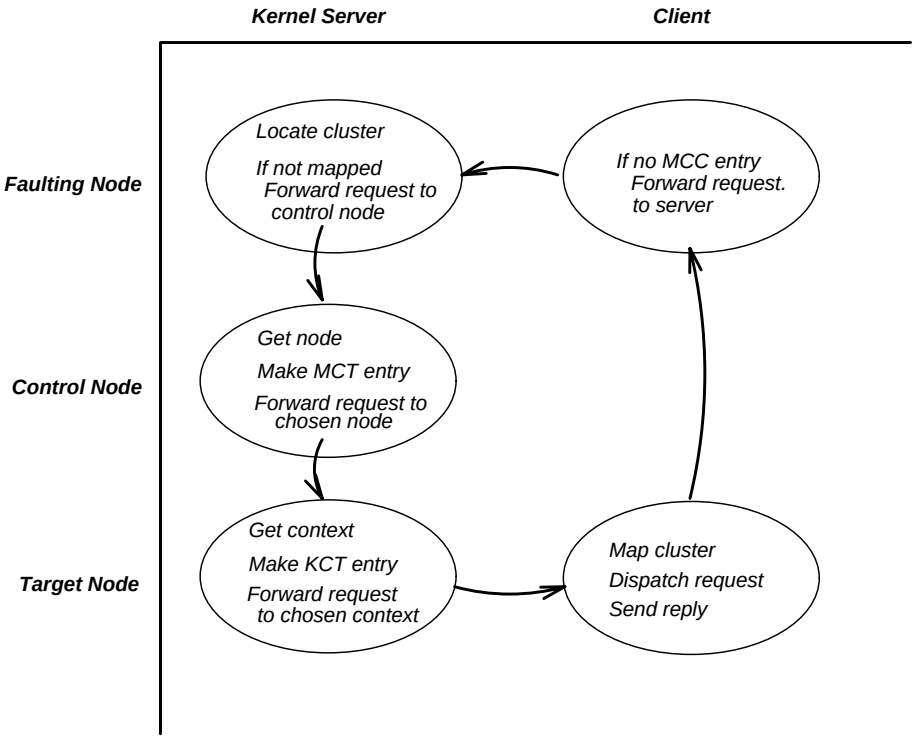

Figure 12: Cluster fault handling when cluster is not mapped.

the requesting node. At the requesting node, the kernel CFH informs the LS before returning to the GRT CFH which then calls the ClM to map the cluster.

1.9 Address Resolution

The main purpose of the ARM is to translate from the name of a local context to its address. To support this function the ARM maintains a register of local contexts which is also used to support the PS (c.f. Sect. 1.5.5).

1.9.1 Data Structures.

At each node the ARM maintains a table – the Address Translation Table (ATT) – mapping from context names to the addresses of the corresponding contexts. The ATT acts as a register of local contexts and contains an entry for each local context.

1.10 The Object Manager

This section presents an overview of the functionality of the OM component of Amadeus. The OM provides support for the management of global and persistent objects building on the cluster abstraction supported by the ClM. Rather than putting much of the burden for doing this on the compiler (or programmer), the OM performs most of the actions required in a language independent manner. Whenever language specific information or actions are required, the OM makes an up-call to code supplied for the language. In particular, the OM provides support for:

- object creation and naming;
- detection of objects faults;
- mapping and unmapping of objects;
- marshalling, unmarshalling and dispatching of invocation requests;
- support for atomic objects;
- clustering;
- garbage collection.

Each of these aspects is briefly summarised below.

1.10.1 Objects.

Fundamentally the OM is concerned with the management of global and persistent objects. From the OM point of view such an object is an opaque element of storage which can be uniquely identified and to which code implementing the interface to the object can be bound dynamically. The OM does not know anything a priori about the internal structure of a particular object nor about the semantics implemented by that object. Information concerning an object required by the OM in order to manage that object must be provided by the language level as described below.

A global or persistent language object – such as, for example, an individual C** or Eiffel** object – must be mapped in a way specific to its programming language, onto an OM object. The most natural mapping is a single OM object for each heap allocated language object. Other mappings are possible. For example, in C**, a language object may be embedded within another language object which is, in turn, mapped onto a single OM object. The term *object* is used as an abbreviation for an OM object, and programming language objects are always qualified as *language objects*.

New global or persistent objects are created by explicitly calling the OM. The OM allocates space for the object (including also space for a header for the object) and returns the address of the object to the language level. When initially created an object is *immature*. Immature objects exist and are known only within the context in which they were created. Objects may become known outside of their context of creation, either because a reference to the object has been passed out of the context, or because the object itself has migrated out of the context. If this happens, the object is *promoted* to being a *mature* object and is given a system wide unique identifier.

An object's header stores information used by the OM to manage the object and to link the object to the code implementing the language specific up-calls required by the OM (c.f. Sect. 1.10.7). In normal operation an object's header is transparent to the language level.

1.10.2 Object References and Object Fault Detection.

A *stub* is a type of object reference, holding enough information not only to locate the referenced object (i.e. its unique identifier) but also, in the case of a reference to a global object, to create a proxy (see later) for it.

On disk a persistent object is stored with its header being followed immediately by the object's data, possibly containing references to other objects, which is followed by a set of stubs for referenced objects.

The layout of an object's data is controlled by the language level and in general, the space allocated by the language for an object reference is too small to store a full stub. Hence when an object is on disk each reference within an object's data is stored as an offset either to the referenced object if stored in the same cluster, or, in the case of an inter-cluster reference, to a stub for the referenced object. This allows the layout of the object's data to be the same as it would be for a "normal" version of the object.

When two objects are co-located in the same context, the format of references between them is determined by the associated programming language. For example, two co-located C** language objects (in two different OM objects) can use direct memory pointers between them for the duration of their co-residence. The OM, with the aid of language specific support, is responsible for *folding* and *unfolding* stubs to and from language references as necessary. Each object carries representation information at run time which allows, amongst other services, the locations of the object references within it to be found.

When an object is fetched into a context, each of its references is examined in turn. If a reference refers to an object which is located in the same context, it may be replaced by a language reference to that object. Subsequent de-referencing of (and invocation via) that object reference can then use the native language mechanism: for example, two co-located C** objects will then use the usual C++ invocation mechanism, and under these conditions will not use the services of the OM. If, on the contrary, a reference refers to an object which is not (currently) located in the same context, some mechanism to trap attempts to access the absent object through that reference must be used. There are a number of possible approaches to trapping attempted accesses to absent objects. One solution is to require an explicit test for the presence of the referenced object, prior to use of the reference. For efficiency, such explicit testing is usually undesirable. An alternative technique, is to instead represent the referenced object in its absence by a *proxy*. Two different forms of proxy are supported by the OM.

A *G proxy* for an absent global object is essentially an OM object which contains no data but is the same size as the object it represents. The code bound to the proxy, which must be provided by the language level, must implement the same interface as the absent object. G proxies are dynamically created by the OM as required. Such proxies need never be stored, and are invisible to application programmers. A reference to a local proxy is unfolded in the same way as a reference to a co-located object. Further, an invocation via that reference will proceed as indicated above using the language mechanism. The proxy is responsible for reacting to the attempted invocation by calling the OM to handle the *object fault*. If the object represented by the proxy is subsequently mapped into the same context the OM will *overlay* the proxy by its principal – in this way any unfolded (language) references to the proxy or to its internal parts from elsewhere in the context remain valid.

A *P proxy* represents an absent cluster containing a (non-global) persistent object to which a reference has been unfolded. A P proxy is a read-protected area of virtual memory large enough to hold the absent cluster should it be mapped into the context. An unfolded reference to an absent persistent object points at the location in memory where the object will be loaded when the cluster is mapped. An attempt to access an absent persistent object will be caught by the OM as a protection violation resulting in an attempt to map the cluster. When the cluster is mapped the access can then be allowed to proceed as normal.

1.10.3 Dispatching and Marshalling.

Dispatching of invocations is performed in a language specific way. In the case of two co-located objects compiled by the same compiler, invocation need not use the services of the OM and so can proceed directly as explained above. For cross-context invocations, each (global) object must be prepared to receive an invocation request in a canonical (Amadeus defined) format and dispatch it in the appropriate language specific way to the appropriate operation or method. Likewise the outcome (if any), including abnormal or exceptional conditions (if any), must be returned and converted into a canonical format. In the case of C** this dispatching mechanism is implemented as an automatically generated additional member function of each class.

Marshalling of invocation frames is the responsibility of both G proxies on the initiating side and of the dispatching mechanism on the recipient side. The OM provides a suite of marshalling routines which allow the language level to marshal the parameters and results of an invocation into the standard message format expected by the kernel including encoding and decoding of individual marshalled values into the canonical format used for transmission between heterogeneous nodes.

1.10.4 Atomic Objects.

The OM does not distinguish between atomic and non-atomic objects. However it does provide a number of routines which are used by the TS in managing atomic objects. In particular, the OM provides a routine to be up-called by the TS to create a recovery point for an atomic-object when necessary. Recovery points must be known to the OM as they must be visible to the garbage collector. In particular, if a recovery point contains the only reference to an object, that object must be preserved by the garbage collector in case the recovery point is restored as the actual state of an object after a transaction abort. The OM also provides the routines to restore an object from a recovery point in case of transaction abort, delete a recovery point in case of transaction commit and to prepare an object to be written to the log (which requires that an image of the object with all of its references folded be created) during the commit protocol execution.

1.10.5 Cluster Management.

It is expected that individual objects are rarely required to be remotely usable, and that it is more likely that a number of objects be grouped together as a single unit of distribution.

In practice many applications do not choose to explicitly manage clusters, but rather employ the default mechanisms of the OM. By default, the OM creates new clusters as required, and places each new object into the most recently created cluster, along with other recently created objects.

As well as the default mechanism, the OM provides primitives to explicitly create a new cluster into which subsequent new objects will be placed; and to move an object into the same cluster as another object.

1.10.6 Garbage Collection and Cluster Unmapping.

The OM currently includes a simple non-incremental mark and sweep garbage collector for immature objects. The roots for the collection are all mature objects within the context and the stacks of all activities executing within the context. Activities within the context are temporarily suspended during garbage collection.

A cluster is naturally unmapped from a context when that context is no longer required. Moreover it is clear that in general, a cluster cannot be unmapped prior to the deletion of the context, unless there are no unfolded references referencing any of the objects within that cluster. To determine if there are unfolded references to an object, the OM must scan the objects in the cluster, and the stacks of all activities executing in the context. This is expensive, and so is coupled with garbage collection of immature objects.

However the OM also provides a primitive to explicitly unmap a cluster from a context. In general, this primitive should be used with care, since for example there may be outstanding invocation frames on the stacks of various activities in the context which have temporary pointer values into the cluster. Nevertheless, an object may be migrated between two machines by explicitly unmapping its cluster at the sending node, and faulting the cluster at the recipient node.

1.10.7 Using the Object Manager.

In interfacing a programming language to the OM, the OM imposes a number of constraints. First, the OM supplies routines to allocate and deallocate global and persistent objects, which should be used in place of the more usual calls to `malloc` and `free`. The OM also provides a number of routines to, for example, aid marshalling, and to test for the presence of an object or to fault on an absent object.

More importantly, the OM expects that the language-specific run-time supplies a small number of up-call routines which the OM can itself call. These in particular include an up-call to dispatch an incoming invocation request, in the canonical format, to an object; to locate object references within an object; and to prepare an object for use. The latter is used, for example within a C** environment to establish any internal virtual table pointers, or virtual base class pointers, within an object.

Further details concerning the interface between a language-specific run-time and the (OM component of the) GRT can be found in Chap. 11. of [Cahill et al. 1993] .

1.11 Cluster Management

Clusters are the basic unit of storage and mapping supported by Amadeus. Clusters are intended to be used by the OM to store groups of related objects. Other components of Amadeus are not aware of which objects are stored in which clusters.

1.11.1 Cluster Structure.

Each cluster has a system wide unique name which is an Amadeus unique identifier. A cluster consists of an integral number of pages. Pages of a cluster are mapped contiguously into virtual memory and appear contiguous when stored on secondary storage.

Every cluster has a small fixed size header known to the GRT. The remainder of the cluster data can be used by the OM for storage of objects and any associated management information needed by the OM to locate objects within the cluster.

1.11.2 Cluster Creation.

Clusters are normally created by the OM in virtual memory. In creating a cluster the OM first allocates a unique identifier for the new cluster by calling the SS. The OM is responsible for allocating space for the cluster in virtual memory and performing any OM specific initialisation. The OM then registers the new cluster with the ClM. The ClM in turn registers the cluster with the LS making the new cluster visible throughout the system.

A new cluster belongs to the extent in which it is created. However, the mapping for a new cluster is not entered into the PS's Cluster Register until the cluster is being unmapped.

1.11.3 Cluster Mapping and Unmapping.

At any time a cluster can be mapped into at most one context anywhere in the system. Clusters are normally mapped as a result of a cluster fault although there is also a primitive available to explicitly request that a cluster be mapped into the current context (where allowable) (c.f. Sect. 1.5.5).

The ClM provides routines to map a cluster into the current context and unmap a cluster from the current context. When mapping a cluster the ClM first determines the size of the cluster by calling the SS and then up-calls to the OM to allocate space for the cluster. The responsibility for all local heap management is assigned to the OM in order to facilitate the implementation of a local garbage collection scheme in the OM. The ClM then reads the cluster from secondary storage into the allocated space and up-calls the OM again in order to allow OM specific initialisation of the cluster.

Clusters are normally expected to be unmapped only when the context into which they are mapped is being deleted (c.f. Sect. 1.6.3). However the interface provided by the CM to unmap a cluster is general and can potentially be called at any time. The ClM is called giving it a pointer to the cluster in virtual memory and the size of the cluster. The ClM writes the cluster to secondary storage before up-calling the OM to release the heap space allocated to the cluster.

1.12 Job and Activity Management

In this section the management of jobs and activities, including job and activity creation, diffusion, termination and control, is described.

Jobs and activities are intended to be provided as first class objects to application programmers[9]. Hence it is expected that a language level object representing each job or activity exists to which language code is bound so that operations on jobs and activities can be invoked using the usual language calling sequence. The AM then provides a more primitive interface for job and activity management which can be called by the language class code and which operates on the separate distributed representation of the job/activity.

Recall that a job is simply a collection of activities where each activity is essentially a thread of control which may span several contexts on the same and/or different nodes. Each activity in turn consists of one or more *processes*[10] where each process is a thread of control executing within a single context. Control within an activity

[9] This is a matter for the language designer in the final instance.

[10] These should not be confused with UNIX processes.

is passed from one context to another by means of cross-context invocations. When a cross-context invocation occurs, the activity's current process is blocked awaiting the results of the invocation and a new process created for the activity in the target context. Thus at any time only one process of an activity may be runnable. The activity's stack is thus distributed over all the processes of the activity with at least one segment of the stack in each context where the activity has an unfinished invocation. An activity can have several processes in the same context if it makes a rebounding call to a context where it already has a (blocked) process. When the activity returns from a cross-context invocation, the process that was created for it in the target context is deallocated.

An activity is said to be *present* in any context where it has at least one incomplete invocation. The activity is said to be *represented* in any context which the activity has visited but at which there are no incomplete invocations. Likewise, a job is said to be *present* at any node where at least one of its activities is present. A job is said to be *represented* at any nodes it has visited but at which the job is not currently present.

1.12.1 Naming.

Jobs and activities are known by so-called `AIds`. An `AId` consists of the concatenation of a `UId` with a context address. The `UId` ensures that the `AId` is globally unique while the context address is used to cache the address of the initial context of the job or activity. The routines to create jobs and activities return `AIds` for the newly created jobs and activities which can then be used in subsequent GRT calls to exercise control over the job or activity. Typically, the `AId` would be cached in a language level object.

1.12.2 Data Structures.

Each job has a descriptor in the job's initial context. The most important pieces of information contained in this descriptor include the job's global state (e.g. running, suspended etc.); a list of all the incomplete activities of the job and the job's so-called 'global lock' which is used to serialise asynchronous operations on the job with activity creations and terminations.

Each activity also has a descriptor in each context in which it is present[11]. The most important information stored in this descriptor includes the `AId` of the job to which the activity belongs; the activity's local state and a LIFO list of the descriptors of the local processes belonging to the activity.

Each context also contains a hash table – the Activity Manager Table (AMT) – which maps the `AId` of a job or activity to the address of the corresponding descriptor in the context.

1.12.3 Job Creation.

Jobs can be created in two different circumstances. The mainline of each application is run as a new job and a running job can also create a new job to perform an object invocation asynchronously. In the latter case the parameter block for the invocation to be carried out is passed as a parameter to the AM routine to create a job.

[11] And in some contexts in which it is represented.

Creating a job only involves allocating a descriptor for the job in the current context with an empty activities-created-list, constructing an `AId` for the job, making a local AMT entry and creating the initial activity of the job.

In the case of asynchronous job creation, the name for the new job is returned to the caller for use in subsequent job control operations.

1.12.4 Activity Creation.

Activities are created either explicitly by an application or implicitly as part of job creation. All activities are created as part of the current job and always in the current context.

Creating an activity is similar to creating a new job: a descriptor is allocated and initialised; an `AId` constructed; an entry made in the context's AMT, and the initial process of the activity created and linked to the new activity. When this process is eventually scheduled it will carry out the target invocation or run the application mainline as appropriate.

Finally, the new activity's name has to be added to its job's list of non-terminated activities. Since the job's list of activities is only maintained in the job's initial context, a message may have to be sent to this context if the activity is being created in any other context. Moreover the activity creation must be synchronised with any asynchronous operations on the job which might be going on in parallel with activity creation. Thus before accessing the job descriptor, the new activity must first acquire the job's global lock, ensuring that no parallel operations are in progress. The activity can then examine the job's global state to determine if any relevant operations have taken place. For example, if the job was explicitly killed during the creation, the activity creation must be aborted or if the job was suspended the activity must be created in the suspended state. If no such operation has taken place the activity is added to the list, so that it will be notified of any further operations on the job, and the global lock released.

1.12.5 Activities and Cross-context Invocation.

On a cross-context invocation a forwarding pointer – in the activity's current process – is left behind giving the address of the target context, if known by the client (c.f. Sect. 1.8). The activity's local state is changed to `remote` and the invocation request message forwarded to the target context. The current process of the activity is then blocked until the reply is received and control within the activity returned to the current context.

On the remote side, the activity may already be represented in the target context and in fact, if the invocation is a rebounding call, it may even be present in the context when the invocation arrives. This is determined by searching the local AMT for an entry for the activity. If the activity is not already represented it is said to *diffuse* to the context and a descriptor is allocated for it and initialised.

In any case, a process is created to carry out the the invocation and is added to the head of the list of processes for the activity in that context. The activity's local state is changed to `active` and the process is scheduled to run.

When the invocation completes, the process will be unlinked from the activity's list and the reply message sent. If the activity has no other processes at the node, then its activity descriptor can be released. However this is not

done immediately, in case the activity performs subsequent invocations at the node in quick succession. Instead the descriptor is allowed to time out. When that occurs the AMT entry for the activity is removed and the descriptor deallocated. The next invocation by the activity arriving in the context will cause a new descriptor to be allocated.

1.12.6 Activity and Job Termination.

An activity terminates when the initial process of the activity exits and thus it always terminates in the activity's initial context. This may happen when the activity's initial invocation completes, when an exception is raised in the activity or when the activity is explicitly killed.

The first step in activity termination is to remove the local AMT entry for the activity prohibiting any further asynchronous operations on the activity. The final step is to inform the owning job about the termination of the activity which may result in a message being sent if the activity was created in a context other than the initial context of its job. The activity is then removed from its job's activities-created-list.

The results of the activity can then be passed back to the creator. For this purpose the kernel up-calls the OM with a pointer to the result block for the activity.

A job terminates when all of the activities belonging to the job have terminated. This is detected during activity termination when the last activity's identifier is removed from the job's list of activities. The job's AMT and descriptor are freed and the results are passed to the OM.

1.12.7 Job and Activity Control.

Users may invoke operations on jobs or activities. The operations available are: suspend, resume, and kill. Operations on jobs effect all activities belonging to that job.

The execution of all operations on jobs is similar. If the operation is requested from a context other than the job's initial context, the request must first be forwarded to the initial context. Once in the job's initial context, the first step is to acquire the job's global lock. This prevents any activity creation or deletion or other job operations from occurring for the duration of the operation. Next the appropriate operation is performed on each of the activities in the job's list of activities. Then the job's state is modified as necessary to reflect the operation just performed. Finally the lock on the job's list of activities is released.

Operations on activities are more complex since they must locate the activity's current process. This is done using a distributed search algorithm. The idea behind the algorithm is to first find a context where the activity is present and then to follow the activity's forwarding pointers to locate the activity's current context. Since the `AId` of the activity is available it is always possible to find one context where the activity is present – its initial context. If no forwarding pointer exists for the activity then it is necessary to wait until control within the activity returns to that context before proceeding with the request. When following forwarding pointers, the most recent process belonging to the activity is always used in order to reduce the number of steps taken. Following the forwarding pointers does not guarantee that the activity's current context will be found. It may happen that at the same time that a request is being forwarded, the activity is returning to the source context. In this case the search must be restarted.

The following shows the steps taken when an activity operation request is processed:

- If the activity is present in the current context and if its current process is located here as well, the operation is carried out.
- If the activity is present in the current context but its current process is located elsewhere, the request is forwarded to the context indicated in the forwarding field of the most recently created process of the activity, if any.
- If the activity is not present in the current context, the request is forwarded to the activity's initial context. The address is in the `AId`.
- If in the activity's initial context and the activity is not present, the search is abandoned. In this case, the activity either never existed or has already terminated.

Once the current process of the activity has been located, the operations proceed as follows:

- **Killing:** Since threads (other than the current thread) cannot be killed (c.f. Sect. 1.16), the activity is marked as killed. Later, the activity must check to see if it has been marked as killed, and if so the activity will terminate itself (this is done by causing the activity's current process to terminate with a fatal exception).
- **Suspending:** As for killing, threads (other than the current thread) cannot be suspended (c.f. Sect. 1.16). Hence, the activity is marked as suspended. Later the activity will check to see if it has been suspended and if so the activity suspends itself on a semaphore located in the activity's descriptor. (Note the check for suspension must be done after the check for killing.)
- **Resuming:** A suspended activity is resumed by signalling on the semaphore used to suspend the activity.

Finally, there is an operation to put the current activity to sleep for a number of milliseconds.

1.12.8 Process Management.

Although a process corresponds to a conventional thread as provided by many other systems, an Amadeus process has more state associated with it and hence the process abstraction is built on an underlying threads package i.e. each process is implemented by a thread.

General FIFO semaphores are the only process synchronisation mechanism provided by the AM for use within the GRT and as a basis for the implementation of other synchronisation mechanisms in the language-specific run-time.

1.13 Load Balancing

The advantages of load balancing in any distributed system are well known, e.g. [Jacqmot et al. 1989]. This section looks at how load balancing is incorporated into Amadeus. The conventional technique of load balancing on process creation used in many systems, [Zhou and Ferrari 1987], maps onto load balancing on activity creation in the case of Amadeus. Furthermore it is also possible to perform load balancing at the granularity of an individual

cluster fault – i.e. load balancing is used by the PS to choose the node at which to map a cluster provided that this does not conflict with the activation policy for the extent to which the cluster belongs (c.f. Sect. 1.5.5.). The justification for this approach is that objects within a cluster exhibit a strong locality of reference and it is likely that the activity will execute within the cluster for a reasonable amount of time before returning to its original cluster or before invoking an operation on another cluster. Long running computations are likely to span multiple clusters each of which will be individually load balanced thus potentially spreading the load of that computation across a number of nodes.

In the current implementation both types (activity and cluster) of load balancing are supported but these are obviously mutually exclusive. When using activity balancing objects being faulted into virtual memory are brought to the activity. When cluster balancing is enabled a placement decision is made on each such fault causing the activity to diffuse if necessary. In either case an activity diffuses to a cluster that is already mapped. It is one of the goals of the current version of Amadeus to experiment with the two types of load balancing to investigate which type is most suitable for our object-oriented applications.

A more in depth discussion of the rationale for this design can be found in [Tangney and O'Toole 1991].

1.13.1 Implementation.

Load balancing is implemented along standard lines, [Eager et al. 1986], with separate load dissemination, load calculation, location policy and transfer policy layers. In fact, load dissemination is handled by the NM (c.f. Sect. 1.3.). Transparent remote execution, often the most troublesome aspect of load balancing, is provided (for free) as an integral feature of the Amadeus environment.

As just outlined, balancing on both activity creation and cluster faulting are supported. It is also possible for the application layer to override the system and explicitly specify (or hint at) the node at which an activity should be created – or an invocation carried out. Finally load balancing can be disabled at system configuration time.

Fundamental to this implementation is the concept of an activity's *preferred node*, i.e. the node at which it will normally execute. If this is not specified by the application at activity creation then it can be chosen by the load balancing module if activity balancing is enabled. Once the preferred node is set all subsequent clusters faulted by the activity are mapped at the preferred node if possible. When cluster balancing is being used a preferred node is not set on activity creation and a separate placement decision is made for each cluster fault. As another alternative the preferred node for an activity can be set explicitly by the application at any time.

Currently load is calculated based on the load average figure returned by UNIX combined with the number of activities on each node. A random element is also used when picking a node in order to avoid swamping and thresholds values are used to avoid going remote when the local node is lightly loaded or the chosen node is already overloaded.

1.14 Transaction Management

The TS encapsulates the RelaX transaction management components. RelaX is described in [Schumann et al. 1989] and [Kroeger et al. 1990]. The integration of RelaX and Amadeus is described in detail in [Mock et al. 1992]. In this section a brief overview of transaction management in Amadeus is presented and the interface between the TS and the other components of the system is described.

1.14.1 RelaX Transaction Management.

In RelaX the active entities in a transaction are known as *participants*. For RelaX, a participant is defined as a thread of control which is confined to a single address space. A transaction may have one or more participants at any time and the set of participants of a given transaction can vary over time. Also in RelaX the units of data accessible within a transaction are known as *resources*.

RelaX supports a model of resources in which each resource has a *committed state* and an *actual state*. The committed state will be updated only within the commit protocol. All accesses to a resource are directed to its actual state, which initially corresponds to the committed state and reflects all modifications to the resource. If a resource is modified within a transaction, a *recovery point* for the resource must be maintained. This recovery point contains enough information to restore the actual state to the before-image of the transaction in case of an abort. Recovery points are value based, but other approaches are not precluded. Note that, as nesting and the use of uncommitted data is allowed, there may be a set of recovery points associated with a resource, i.e. the actual state of the resource is accessible to all active transactions, each saving its individual before-image in a recovery point. In case of a transaction abort the corresponding recovery points are restored.

RelaX is composed of two components, the Transaction Manager and the Generic Transaction Support components. The transaction manager comprises the following components: the recovery graph, the action tree, the commit and abort protocols and the log component, thus subsuming all transaction-specific, resource-independent tasks related to distributed transaction management. The generic transaction support comprises generic recovery and concurrency control supporting resources at a single node. This clear distinction between transaction management and data management corresponds to the X/Open proposal for distributed transaction management [X/Open 1989].

Due to the use of non-strict two-phase locking, dependencies between transactions may arise. The recovery graph is a distributed data structure which on the one hand stores dependencies between transactions and on the other hand keeps track of the nodes visited by a transaction. The recovery graph is consulted during commit/abort protocol execution.

The action tree is a distributed data structure which reflects the nesting structure of transactions and which stores the relationship between transactions and their participants.

The commit protocol is a decentralised agreement protocol that ensures that either all nodes involved in a commit request commit a set of transactions or all these nodes abort the effected transactions. If a node crashes during protocol execution the operational nodes do not have to wait for the faulty node to recover in order to come to a decision that is nevertheless consistent with the view of the crashed node (i.e. a non-blocking protocol). The abort protocol is a decentralised protocol which ensures that an aborted transaction and all of its dependents leave no effect in the distributed system.

The log is a sequence of typed records stored on stable storage with fast sequential access. In the first phase of the commit protocol the effected resources are stably but still revocably stored. Furthermore, the log stores the state of commit requests. The information stored on the log is used during restart. If a faulty node recovers, the log is analysed to select the relevant information.

The generic concurrency control implements non-strict two-phase read/write locking for resources accessed by nested transactions. The generic recovery control manages recovery points of resources. It initiates the creation of

a recovery point if necessary and the restoration of a recovery point in case of a transaction abort. Furthermore, during processing of commit requests it determines the correct image of the resource to commit.

1.14.2 Amadeus/RelaX Integration.

The following sections give an overview of the main issues that had to be addressed in the integration of Amadeus and RelaX.

Resource Management. The resources managed by the TS are individual atomic objects. As for other objects, every atomic object must belong to some cluster. Moreover, a particular cluster can store both atomic and non-atomic objects. In this section, the term 'atomic object' is abbreviated by 'object'.

The SS stores only committed versions of objects. Transactions operate on objects which are mapped into virtual memory. A given object may be accessed by many different transactions while mapped into virtual memory. An object can only be unmapped and written to secondary storage when no transactions are using it. Transaction management is therefore orthogonal to the mapping and unmapping of objects, and in particular, the SS is not involved in committing updates to objects.

Before a mapped object is updated by a transaction, a recovery point is created for the object in virtual memory. If the transaction aborts, the state of the object will be restored to the state given in the recovery point. Before the transaction commits the final state of the object in the transaction is stably saved to prevent its loss in the event of a subsequent failure. For this purpose each context has an associated data log on which a copy of the final state of the object is saved during the prepare phase of the two-phase commit protocol. If the transaction commits, any recovery points for objects modified by the transaction can be discarded. At that point the committed state of the object is still mapped into virtual memory. The copy of the object contained in the SS (if any) is now obsolete but will be updated once the object is unmapped. In the interim, all accesses to the object will be directed to the object mapped into virtual memory. The object may subsequently be modified and changes to it committed by other transactions before being unmapped.

If the node or context in which the object is mapped fails before the object is unmapped, the copy of the object in virtual memory is lost. The up-to-date copy of the object now exists only in the context's log. In this case, even if the node is down, the object, and indeed the cluster to which it belongs, is still considered to be mapped at the node although it will be unavailable until the node recovers (c.f. Sect. 1.7).

On a context failure, the context will be restarted and its log used to obtain the youngest committed version of each object that was mapped into the context. Although the unit of transfer between the SS and virtual memory is a cluster, the unit of logging is an object. Hence when recovering a context, the TS must read the clusters, to which the objects found in the log belong, from the SS and overwrite those objects, to which changes have been committed, with the copy of the object obtained from the log. After recovery, the context will therefore contain only those clusters which were present before the crash and which had contained objects to which changes were committed since the cluster was mapped. Other clusters which were present in the context before the crash are lost from virtual memory (c.f. Sect. 1.7).

After a node failure, the local TM restarts each context that existed before the crash and that was involved in a transaction.

When a cluster is unmapped from virtual memory, it is necessary to write a record to the log to indicate that the cluster need not be recovered in the context. In addition, in order to reclaim log space and reduce restart time in the case of failure, it is possible to write a copy of a cluster containing only committed updates back to the SS without unmapping the cluster. In this case a record is forced to the log to delimit the extent of log records which must be applied to the cluster after restart.

Participant Management. In Amadeus, each participant corresponds to a process as defined in Sect. 1.12.8.

When an activity begins a new transaction, a new process is created for the activity and becomes the initial participant of the transaction. The process which was running when the transaction was created is blocked until the transaction completes. Creating a new process as the initial participant of each transaction allows the state of the activity to be rolled back cleanly in the event of the transaction aborting i.e. simply by deleting all participant processes.

Participants are added to a transaction whenever a new activity is created within the transaction or when any activity involved in a transaction performs a cross-context invocation. Such participants must be registered with the TM at the node where they are created and unregistered as they terminate.

Additionally whenever a participant makes a remote call the TM at both the sending and receiving node must be informed of both the call and the return. A certain amount of control information is also attached by the TM to all remote call and return messages sent within a transaction (c.f. [Schumann et al. 1989]). Moreover, when an activity involved in a transaction performs a cross-context invocation, it cannot return from that call until all processing initiated on behalf of the call has completed. In particular, this means that the activity must wait for all activities created as a result of the call to complete.

Finally, if a cross-context invocation results in a new context joining the transaction, the TM must be informed.

1.14.3 The Transaction Sub-system Interface.

The main interface to the TS is that used by a language (via the OM) to manage transactions and, in particular, to make attempts to access atomic objects known to the TS.

The TS uses the services of the OM to make recovery points for individual atomic objects and also to prepare atomic objects for writing to the log.

A number of routines are provided for participant management which are used by the AM to inform the TS when new participants are created and deleted. The TS also provides *synchronisation points* as a means of ensuring that all processing on behalf of a cross-context invocation is complete when the call returns. A synchronisation point is essentially a count of the number of participants which are active on behalf of the call in the target context. Each call results in the creation of a new synchronisation point. When a local participant is created, the count is incremented. When the participant exits the count is decremented. The call cannot return until the count goes to zero.

Finally, the TS provides routines used during remote invocation to inform the TMs at the sending and receiving nodes of the call and the return.

1.15 Communications

The CS handles all communication that takes place in an Amadeus system. Depending on the request, one of a number of different possible modes of communication can be used.

- ordered atomic multicast;
- asynchronous notification;
- synchronous request-reply;
- forwarding.

The CS must support each of these forms of communication. Moreover, an important design goal was to make the other components of Amadeus totally independent of the implementation of the CS. For these reasons the high level interface to the CS is cast in terms of the remote requests that can be initiated – i.e. with one routine for each remote request that can be made. Hence, the interface to the CS seen by the kernel and by the GRT is entirely different.

Our intention in defining the interface to the CS in this form was that the CS calls could be implemented using an existing RPC package. Indeed, a prototype version of a CS (excluding atomic multicast) using SunRPC has already been implemented. However, such an implementation is limited by the functionality of the underlying package – for example message forwarding is not typically supported and thus has to be built on top.

For these reasons a message transport service – the Inter-Kernel Message service (IKM) – which provides the various modes of transmission required by Amadeus on top of an underlying unreliable connectionless transport service was implemented.

The IKM provides for the transmission of arbitrary sized messages between clients and kernels in either request-reply or request-only mode. In addition, the IKM supports the forwarding of a message received previously. Finally, the IKM provides the interface to the ordered broadcast service.

The ordered multicast mechanism is provided by RelaX. The protocol used is based on that described in [Chang and Maxemchuk 1984] and is fully described in [Vonthin 1987]. It is not discussed further here.

The other services are supported by a lightweight protocol similar to that used in the V system [Cheriton 1983], but with selective retransmission of partial messages, which is built on top of the unreliable UDP datagram service but which has been designed to be independent of the underlying transport layer.

1.15.1 Structure of the Communications Sub-system.

Whether initiated by the kernel or GRT, all remote requests follow a similar path.

The caller (a GRT or kernel component) calls a stub routine for the particular request which it wishes to make. In general, the parameters to this routine depend on the particular request. However all requests take as parameter the address of the callee. Typically this will have been obtained by the caller from the ARM or as a result of a

previous request. The role of the stubs is essentially to marshal the parameters of the request into a message, encoding data in a canonical form suitable to support transmission between heterogeneous nodes when required.

In the case of messages including an invocation request i.e. a `tblock`, the encoding and decoding of the `tblock` body is the responsibility of the language-specific run-time which is the only component of the system with the type information necessary to interpret user requests. Typically this knowledge is encoded in a language level stub. The `tblock` body is not interpreted by the CS stub routine but is included in the request message as opaque data. The `tblock` header is of course encoded appropriately.

Currently the stubs are hand coded. However an implementation using a standard RPC package could also make use of a stub compiler to generate the stubs from an appropriate IDL description of the interface.

The message is then sent using the underlying message service. Depending on the request the message can be sent synchronously, in which case the caller will be blocked until a reply is received, or asynchronously when no reply is required.

Requests are executed as they arrive in the target process and in parallel with other requests, using threads. On the callee's side, a thread is created for the incoming request and a stub routine for the request is called to decode the message. The appropriate component is then called to carry out the request. If a reply is required it is initiated explicitly by the called component.

1.16 Thread Management

Threads are the basic building blocks for processes in the GRT and are used to handle parallel requests in the kernel. Each process is implemented by a single thread. Threads are also used for other Amadeus system functions.

The goal was that various thread packages (including our own implementation) may be used with minimal change to the existing code. For this purpose a generic thread interface has been designed which defines the minimum functionality needed by Amadeus from a threads package.

The most basic operations defined on threads are operations to create a thread and to exit from a thread. When creating a thread, a pointer to a function is used to specify the starting point for the thread. If the thread returns from this function, the thread is terminated. Threads are given priorities which are only taken as hints (i.e. they may be ignored).

Other miscellaneous operations defined are: an operation to put the current thread to sleep for a number of milliseconds, an operation to allow the current thread to give up the processor allowing another thread to run, and an operation to implement a non-blocking select.

There are a number of limitations which result from the thread interface. These are:

- A thread (other than the current thread) cannot be killed. This means that when killing an activity (c.f. Sect. 1.12.7) the activity must be marked and later volunteer itself to be killed. This potentially means that an activity might never terminate if it never makes the check to see if it should do so.
- A thread (other than the current thread) cannot be suspended. This introduces similar problems as with killing activities when an activity is suspended.

- The OM garbage collector needs access to the stacks of all the threads in the current context. However not all thread packages allow access to the threads' stacks so some other form of garbage collection is required.

2 Other Implementations

2.1 CHORUS

The CHORUS implementation of the Comandos virtual machine – CHORUS/COOL v2 – differs from other work in the project in that the implementation was designed, from the outset, to run above the CHORUS nucleus [Rozier et al. 1988] (often referred to as a micro-kernel) as opposed to above a traditional operating system kernel like UNIX.

The main goal of this work was to understand how new generation operating systems, in particular, those based on the object-oriented programming model could be implemented above the CHORUS nucleus. In particular, the goal was to understand the basic set of mechanisms that must be present at the system level to support such operating systems. To do this, requirements from Comandos and ISA [APM 1991], together with experience from the earlier CHORUS/COOL v1 platform, were taken into account in the design of a generic base level. The generic base level is an extension of the existing CHORUS mechanisms which supports all of these requirements. Two of the expected benefits of building at such a low level are worth mentioning:

- The CHORUS Nucleus is designed to support distributed operating systems and thus provides general operating system abstractions suited to distribution, thus, it was not necessary to add distribution support.
- Because it was not built above an existing operating system, the design of the platform is not constrained by the abstractions offered by such an operating system to application builders – which are rarely those that system builders require – nor hampered by its inefficiencies.

To achieve this goal, the kernel level provides a set of distributed mechanisms well suited to supporting the Comandos GRT (c.f. Fig. 13) above the CHORUS Nucleus.

It is interesting that the COOL implementation runs alongside the existing UNIX (CHORUS/MiX) system. The COOL system provides an object-oriented programming environment that allows access to the UNIX world. In fact, the COOL implementation uses part of the UNIX system implementation, in particular its file system.

2.1.1 The CHORUS System Building Architecture.

The CHORUS Nucleus offers a basic set of mechanisms to support operating system development. *Actors* serve as a unit of naming and resource allocation in the system. *Ports* are attached to actors and provide an end point for communication. Ports are globally named within a set of distributed nodes and the Nucleus is responsible for locating a port and delivering a message to that port. Memory is composed of *regions* attached to actors that can be backed by a secondary storage notion referred to as a *segment*. The relationship between regions and segments is managed by a *mapper*, which is external to the Nucleus. This allows mappers to be system specific – implementing a particular policy for each system. Lastly, *threads* are provided by the Nucleus and can be attached to a particular actor.

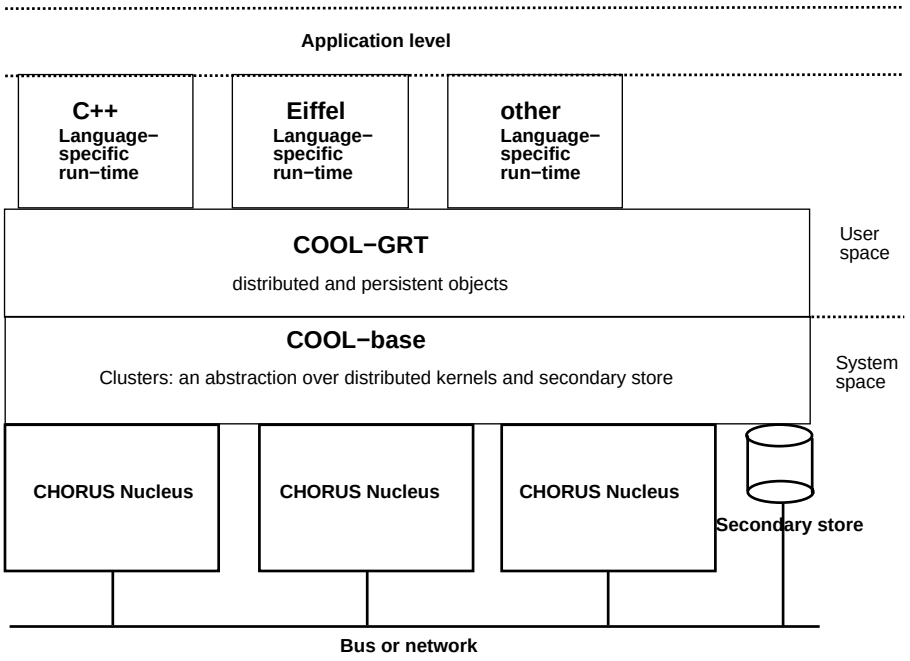

Figure 13: COOL architecture.

Distributed operating systems are built above the nucleus by implementing the functionality of the operating system in a set of independent servers. Each server is a system actor and interacts with other actors to provide a global, distributed operating system interface. For example, CHORUS/MiX V4, a binary compatible implementation of the UNIX SVR4 system is composed of five actors, a process manager that acts as an interface to the system and manages processes, an object manager that implements the file system, a streams manager, an IPC manager for System V IPC and a key manager for distributed file coherency.

2.1.2 Extending CHORUS to Support Object-oriented Systems.

Although it has been proved that the basic Nucleus services are well suited to traditional operating systems, even those extended to support transparent distribution, experience with the CHORUS/COOL v1 system had proved that some extensions to these mechanisms were necessary to support object-oriented systems. The COOL-base level incorporates these extensions and provides an extended system interface.

The major extension that the COOL-base level supports is a distributed notion of persistent virtual memory based on a model called the *persistent context space*. The model supports distributed memory through *persistent contexts* and *context spaces*. A persistent context supports data which is guaranteed to persist. A context space is a collection of persistent contexts whose mapped data is maintained in a coherent manner.

The internal structure of a persistent context is based on the notion of *containers* and *clusters* (these are COOL-specific terms and should not be confused with the Comandos terms). A cluster represents a portion of virtual memory supporting one or more GRT level objects. A container is a grouping of clusters such that memory is complete i.e. all direct memory references are resolvable within the container.

Coherency between clusters in a context space is managed by the COOL mapper and uses a distributed, strict coherency mechanism, as described in [Li and Hudak 1989]. Allocation of memory across context spaces uses a global allocation mechanism provided by the COOL-base level.

Clusters are the unit of mapping and unmapping. A cluster, when requested by the GRT level will be located and if necessary mapped into a persistent context. Thus the base level supports a single level store abstraction, providing persistence as required by the Comandos virtual machine, and extends this with an implementation of distributed virtual memory that is suited to object-oriented systems like Comandos.

The base level is implemented as a collection of CHORUS system actors, which, for efficiency reasons, are loaded into the Nucleus address space and accessed via a trap interface. Each COOL-base actor uses the CHORUS IPC mechanism to communicate with other COOL-base actors and to support the distributed management protocols.

2.1.3 The COOL Generic Run-time.

The COOL GRT provides the basic set of Comandos virtual machine services (although there is currently no support for the TS and only a lightweight version of the PS) and uses the base level functionality.

The ES is based on the micro-kernel supported threads. Each activity is mapped to a thread, with one thread acting as a virtual processor and possibly supporting multiple Comandos activities. Each activity is uniquely named and managed by a global job management scheme that uses CHORUS IPC to maintain global state.

The GRT uses the cluster abstraction provided by the base level as a place to create and manipulate objects. Thus the COOL version of the Comandos VOM is implemented by both the base level and part of the GRT level. Each cluster comprises two zones, a GRT zone where cluster specific data is held and a user zone where user objects are placed. Clusters are named using a port identifier. Thus for each cluster that exists within a context the GRT creates a port. Communications directed to that cluster are received by the GRT managing that cluster at the port. Objects are named relative to a cluster such that invocation of an object at the GRT interface is mapped into a message sent to the cluster in which that object resides. Since port names are globally unique, a cluster and thus objects managed by the cluster can be moved around the distributed system as required. Hence, the implementation of the Comandos object invocation mechanism is greatly simplified.

As discussed in Chap 3. of [Cahill et al. 1993] support for mapping objects into and out of address spaces, using pointer swizzling, based on language-specific information to allow relocation, is provided. The unit of mapping in the COOL implementation is the cluster. Each cluster, comprising multiple virtual memory regions, is mapped by an equivalent number of storage segments. When a cluster is mapped to or from store, the base level reads or writes regions using the COOL mapper. This operation can be likened to a paged virtual memory system where pages of data are similar to regions. The COOL mapper is responsible for managing both the consistency of the data across contexts and the consistency of the data on secondary store. Thus, the COOL mapper (or mappers) collaborate to provide the SS. This is actually based on the UNIX file system and uses files to hold segments.

2.1.4 Conclusion.

The CHORUS implementation of the Comandos architecture was not designed to provide yet another implementation of the same interface over UNIX, but to explore the use of the CHORUS micro-kernel as support for a

system-level implementation of Comandos. In all cases the basic micro-kernel mechanisms have been used to support global management schemes that are scalable. In general, the basic system model, and mechanisms that CHORUS supports are adequate to support a system such as Comandos. However, the virtual memory system has been greatly extended to support objects in a distributed environment allowing increased flexibility and performance to be gained.

The COOL programming environment consists of a tool (COOLPP) that is capable of generating multiple pre-processors, and a COOL++ preprocessor that implements the C++ language-specific run-time.

The system has been validated using the Comandos pilot application, CIDRE. A fully functional version of CIDRE, written in C++ currently runs on a network of Intel 386 machines, hosting CHORUS/MiX and the COOL implementation of the Comandos virtual machine.

2.2 Mach

The Mach micro-kernel [Accetta et al. 1986, Golub et al. 1990] was developed at Carnegie Mellon University, as a new foundation for operating systems. It provides extensible memory management, threads, and an extensive IPC facility.

In versions 2.5 and earlier, Mach was combined with UNIX to deliver a complete operating system environment; the architecture of OSF/1 is similar to Mach 2.5. Version 3, however, provides Mach as a pure kernel, with no other operating system functionality in kernel space. Particular operating system environments are provided by means of user space servers. To date, servers have been prototyped for BSD, OSF/1, SVR4, Sprite, and DOS.

Although it is a goal of the Mach kernel to minimise abstractions provided by the kernel, it is not a goal to be minimal in the semantics associated with those abstractions. As such, each of the abstractions provided has a rich set of associated semantics. Although this makes it difficult to identify key areas, the main kernel abstractions are considered to be the following:

- Task – the unit of resource allocation encompassing address space and port rights.
- Thread – the unit of CPU utilisation.
- Port – communication channel, accessible only via send/receive capabilities (rights).
- Message – collection of data objects.
- Memory object – internal unit of memory management

2.2.1 Goals of the Mach Implementation of Comandos.

The micro-kernel approach allows other servers to be developed, as well as those implementing the semantics of an operating system. However, until recently, there has been very little work on implementing servers over Mach, other than those implementing different operating system personalities. The basic mechanisms offered by Mach can be matched closely to the abstractions of the Comandos virtual machine, and this provides the motivation for implementing Comandos over Mach, with two major objectives:

- To evaluate the benefits of the micro-kernel technology for the support of a distributed object-oriented operating system such as the Comandos virtual machine;
- To investigate the suitability of Mach for application servers other than those emulating operating system personalities, with particular emphasis on performance evaluation.

Within the Comandos project, two approaches have been explored: adaptation of existing UNIX implementations to use the Mach mechanisms (this approach has been followed both in adapting Guide-1 to run over Mach 2.5 [Boyer et al. 1991] and in adapting Amadeus and IK to run over Mach 3.0; this approach is described in Sect. 2.2.2 below. A more fundamental approach involves a major redesign based on the exploitation of the Mach 3.0 capabilities, and this approach has been adopted for the Guide-2 implementation, which is described in Sect. 2.2.3.

2.2.2 Adaptations of Existing Comandos Implementations.

These adaptations implemented only a subset of the Comandos virtual machine, and were carried out essentially as feasibility studies, to test the ideas of mapping Comandos abstractions onto Mach ones. For example:

- the use of Mach tasks for Comandos jobs;
- the use of Mach threads for Comandos activities;
- the use of Mach ports as object references.

Both the IK and Amadeus adaptations to Mach 3.0 continue to use UNIX functionality in addition to the Mach mechanisms. This is provided by a user-level UNIX server (work has been done using both the BSD UNIX server, developed by Carnegie Mellon University, and the OSF/1 server, developed by the OSF Research Institute).

The modifications to Amadeus in adapting it to run over Mach 3.0 included:

- the use of Mach Cthreads to replace the Amadeus non-preemptive lightweight process package based on the interval timer;
- the use of the Mach network shared memory server for shared information within one host (UNIX Amadeus uses System V shared memory);
- UNIX Amadeus uses NFS for the SS; the Mach version uses either AFS or the Mach Remote File System;

The current version continues to use UDP datagrams as the basis of the IKM protocol, and has not been modified to use Mach IPC.

In MachIK, the version of IK adapted for Mach 3.0, a port is associated with each distributed object, and send rights are used as references to these objects. Using ports as distributed object references has several advantages:

- Location transparency – a message can be sent to a port in a location transparent way. This built-in Mach feature provides a cheap way to implement location transparent invocation.

- Port migration – receive rights can be moved between tasks. When migrating an object the receive right is moved to the destination context; the other contexts can continue to invoke the object unaware of its movement. However, the kernel does not yet properly support the migration of receive rights.
- No more senders notification – the programmer can request a notification when the last send right to one of his receive rights dies. This basic mechanism was used to implement garbage collection for distributed objects.
- Protection by capabilities – access to objects is protected by the capability mechanism associated with Mach ports.
- Port death notifications – if requested, a task can receive notification whenever one of the rights in the port name space dies. This mechanism is useful in detecting child death and orphans in remote invocations.

Lazy evaluation techniques are used to reduce the overhead associated with distributed objects. A port is only allocated when a reference to the related object is exported.

The current implementation does not exploit the user level memory management mechanisms provided by Mach which offer interesting possibilities such as the ability to share objects using distributed memory.

2.2.3 Native Implementation of Comandos on top of Mach 3.0.

This section describes the principles of Guide-2, a native implementation of the Comandos system on top of the Mach 3.0 micro-kernel.

As for Amadeus and MachIK, jobs and activities are mapped onto Mach tasks and threads. More precisely, any context within a job is represented by a Mach task. This scheme provides isolation of context address spaces. Consequently, a job is a collection of Mach tasks, one per context.

A major feature of Guide-2 is the extensive use of memory management facilities offered by Mach 3.0, i.e. *external pagers*. External pagers allow the simultaneous implementation of several strategies for sharing objects over a network: a single image for an object or multiple copies of an object with different consistency policies. The main characteristics of the Guide-2 VOM are summarised below; details of the design can be found in [Boyer 1991].

- A cluster is represented by a Mach memory segment which is handled by a specific external pager. Class objects (i.e. methods code) are shared using a "multiple image" policy while data objects are shared using the "single image" policy.
- Contexts are represented by Mach tasks, and a Mach port is associated with each context. This allows:
  - high-level protection, as port identifiers are unforgeable, and thus it is not possible to access any object within a context without permission ("permission" here being assimilated to the port identifier).
  - migration of contexts, as ports are themselves subject to migration. This facility is used to support load balancing.
  - handling of node failures, as event notification on ports allows the detection of abnormal termination of jobs and activities.

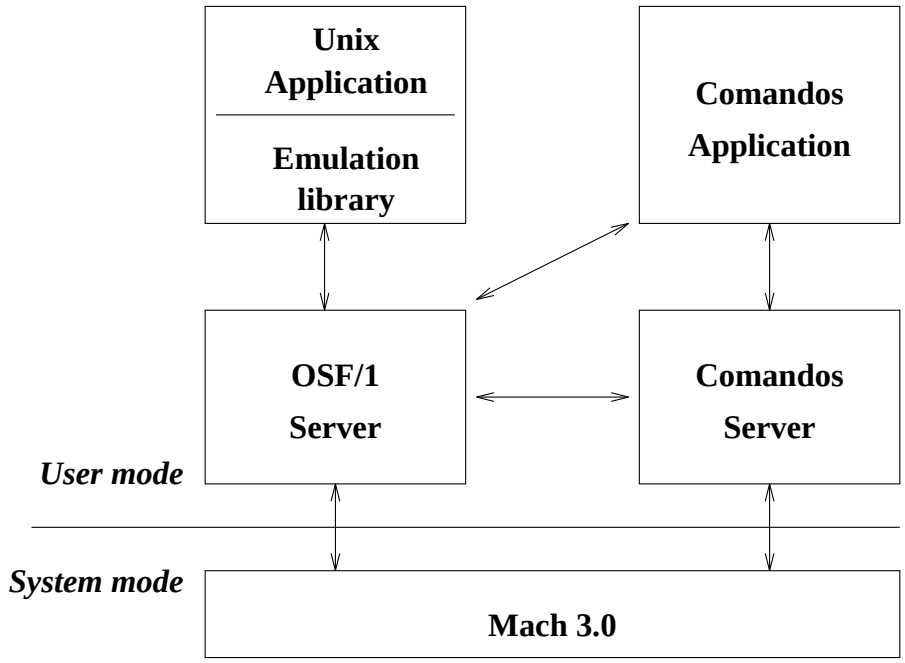

Figure 14: Guide-2 architecture.

Guide-2 is implemented on a network of i486-based Zenith machines, running OSF/1 MK. The general architecture is depicted in Fig 14. On top of Mach 3.0 two servers coexist: the OSF/1 single server which implements the UNIX environment, and the Comandos server which implements the Comandos virtual machine. The Comandos server, as well as Comandos applications, can reuse all of the services provided by the UNIX environment. In the current version only the Comandos language is supported by the Comandos server.

References

[Cahill et al. 1993] Cahill, V., Balter, R., Harris, N. and Rousset de Pina, X. (Eds.) *The Comandos Distributed Application Platform.* Springer-Verlag, Berlin, 1993.

[Accetta et al. 1986] Accetta, M., Baron, R., Bolosky, W., Golub, D., Rashid, R., Tevanian, A. and Young, M. Mach: A New Kernel Foundation for UNIX Development. In *Proceedings of the Summer USENIX Conference*, Atlanta, GA, USA, 1986. USENIX, Berkeley, 1986. pp. 93–112.

[APM 1991] Architecture Projects Management Ltd. *ANSA: An Engineer's Introduction to the Architecture.* Architecture Projects Management Ltd., Cambridge, 1991.

[Boyer 1991] Boyer, F. A Causal Distributed Shared Memory Based on External Pagers. In *Proceedings of the USENIX Mach Symposium*, Monterey, CA, USA, 1991. USENIX, Berkeley, 1991. pp. 41–57.

[Boyer et al. 1991] Boyer, F., Cayuela, J., Chevalier, P.Y., Freyssinet, A. and Hagimont, D. Supporting an Object-Oriented Distributed System: Experience with UNIX, CHORUS and Mach. In *Proceedings of the* 2^{nd} *Symposium on Experiences with Distributed and Multiprocessor Systems*, Atlanta, GA, USA, 1991. USENIX, Berkeley, 1991. pp. 283–300.

[Cahill et al. 1992] Cahill, V., Taylor, P., Starovic, G., Tangney,, B. and O'Grady, D. *Supporting the Amadeus Platform on UNIX*. Technical Report TCD-CS-92-25, Department of Computer Science, Trinity College Dublin, 1992.

[Chang and Maxemchuk 1984] Chang, J. and Maxemchuk, N. Reliable Broadcast Protocols. *ACM Transactions on Computer Systems*, 2(3):251–273, 1984.

[Cheriton 1983] Cheriton, D. Local Networking and Internetworking in the V-System. In *Proceeding of the* 8^{th} *Data Communications Symposium*, North Falmouth, MA, USA, 1983. IEEE/ACM, Los Angeles, 1983. pp. 9–16.

[Eager et al. 1986] Eager, D.L., Lazowska, E.D. and Zahorjan, J. Adaptive Load Sharing in Homogeneous Distributed Systems. *IEEE Transactions on Software Engineering*, SE-12(5):662–675, 1986.

[Golub et al. 1990] Golub, D., Dean, R., Forin, A. and Rashid, R. UNIX as an Application Program. In *Proceedings of the Summer USENIX Conference*, Anaheim, CA, USA, 1990. USENIX, Berkeley, 1990. pp. 11–15.

[HARNESS 1991b] Harness Consortium *Harness Platform: Basic Specification.* Harness/PSE1/DEL/COO/001/1.0, Harness Consortium, 1991.

[Jacqmot et al. 1989] Jacqmot, C., Milgrom, E., Joosen, W. and Berbers, Y. UNIX and Load Balancing: a Survey. In *Proceedings of the EUUG Spring Conference*, Brussels, Belgium, 1989. EUUG, Buntingford, 1989. pp. 1–15.

[Kroeger et al. 1990] Kroeger, R., Mock, M. and Schumann, R. The RelaX Transactional Object Management System. In Rosenberg, J. and Leslie Keedy J. (Eds.): *Proceedings of the International Workshop on Computer Architectures to Support Security and Persistence of Information*, Workshops in Computing, Springer-Verlag, Berlin, 1990. pp. 339–355.

[Li and Hudak 1989] Li, K. and Hudak, P. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 7(4):321–359, 1989.

[Mock et al. 1992] Mock, M., Kroeger, R. and Cahill, V. Implementing Atomic Objects with the RelaX Transaction Facility. *Computing Systems*, 5(3):259–304, 1992.

[Rozier et al. 1988] Rozier, M., Abrossimov, V., Armand, F., Boule, I., Gien, M., Guillemont, M., Herrmann, F., Kaiser, C., Langlois, S., Léonard, P. and Neuhauser, W. CHORUS Distributed Operating Systems. *Computing Systems*, 1(4):305–370, 1988.

[Schumann et al. 1989] Schumann, R., Kroeger, R., Mock, M. and Nett, E. Recovery Management in the RelaX Distributed Transaction Layer. In *Proceedings of the* 8^{th} *Symposium on Reliable Distributed Systems*, Seattle, Washington, USA, 1989. IEEE, Los Alamitos, 1989. pp. 21–28.

[Sousa et al. 1993] Sousa, P., Zúquete, A., Sequeira, M., Guedes, P. and Alves Marques, J. *IK-2 Implementation Report.* Technical Report COMANDOS-INESC-TR-0040, INESC, *To appear 1993.*

[Steiner et al. 1988] Steiner, J.G., Neumann, C., and Schiller, J.I. Kerberos, an Authentication Service for Open Network Systems. In *Proceedings of the USENIX Winter Conference*, Dallas, TX, USA, 1988. USENIX, Berkeley, 1988. pp. 191–202.

[Tangney and O'Toole 1991] Tangney, B. and O'Toole, A. An Overview of Load Balancing in Amadeus. In Ammar, R.A. (Ed.): *Proceedings of the 4th ISMM/IASTED Conference on Parallel and Distributed Computing and Systems*, Washington DC, USA, 1991. Acta Press, Anaheim, 1991. pp. 144–146.

[Vonthin 1987] Vonthin, R. *Spezifikation des PROFEMO-Reliable Broadcast Protokolls in UNIX 4.2BSD*. GMD-Studie 127. GMD, Sankt Augustin, 1987.

[X/Open 1989] X/Open Company *X/Open Preliminary Specification: Distributed Transaction Processing: The XA Specification*. X/Open Company, Reading, 1989.

[Zhou and Ferrari 1987] Zhou, S. and Ferrari, D. A Measurement Study of Load Balancing Performance. In *Proceedings of the 7th International Conference on Distributed Computing Systems*, Berlin, Germany, 1987. IEEE, Los Alamitos, 1987. pp. 490–497.