Amadeus Project

Overview of the Amadeus Project

Distributed Systems Group

Distributed Systems Group
Dept. of Computer Science
Trinity College Dublin

Abstract

An introduction to the Amadeus v1.0 environment for distributed and persistent programming in C++ is described.

Document Identifier A390
Document Status Red Version. Report.
Created 21 December 1990
Revised 8 May 1991
Distribution Public

Contents

List of Figures

Preface

The Amadeus v1.0 release is the first release from the Amadeus project by the Distributed Systems Group of the Department of Computer Science, Trinity College Dublin (TCD). It extends C++ for distribution and persistence; the extended language is called C**.

Amadeus v1.0 is itself implemented in C and C++, on top of Digital Equipment Corporation's Ultrix 3.1 on both MicroVAXes and DECstations. A small amount of assembly language is included to support a lightweight process package. The system relies on the use of NFS for remote file access. The implementation also uses Ultrix sockets, signals, shared memory, and semaphores. The C**compiler in Amadeus v1.0 is a modified version of the Free Software Foundation's g++ 1.37 compiler.

Future releases, both planned and underway, will extend the functionality, supported languages and host operating systems.

The Amadeus Project has been influenced by the Esprit Comandos-1 and Comandos-2 projects, in which TCD has been a partner. The project is also being influenced by the Esprit Harness and Ithaca projects. TCD acknowledges the fruitful interactions with all of the participating institutions and in particular with Inesc in Lisbon; Bull and IMAG in Grenoble; GMD in Bonn; and the University of Glasgow.

Information Set

The information set for the Amadeus v1.0 release includes the following three documents:

- *Overview of the Amadeus Project* presents an overview of both the Comandos and Amadeus projects. This document also includes an introduction to distributed systems and explains the terminology used in the Amadeus Project.
- *C** Programmers' Guide* contains both tutorial and reference information on programming in the C**language. The text assumes a knowledge of object-oriented programming and, in particular, the C++ language. This guide lists a sample C**program which is supplied with Amadeus v1.0 which can be compiled and linked after installing the Amadeus environment.
- *Amadeus Installation and Maintainence Guide* contains the installation procedure

together with instructions for startup, and configuration.

Organisation of This Document

The intended audience for *Overview of the Amadeus Project* is applications programmers with a background in object-oriented programming, but possibly having only a limited knowledge of distributed processing. This text also provides a management overview of both the Comandos and Amadeus projects. The overview is arranged in three chapters:

Chapter 1 introduces the reader to distributed processing and persistence which are both features of the Amadeus environment.

Chapter 2 presents an overview of the Amadeus Project and explains how Amadeus manages objects. It also describes the relationship of the Amadeus Project to the Comandos Project.

Chapter 3 describes the Amadeus environment including the components of this environment and how they interact.

Related Texts

The following texts are recommended as background reading on the C++ Language:

- *The Annotated C++ Reference Manual*, Bjarne Stroustrup, (1990), Addison Wesley.
- *Users's Guide to GNU C++ (version 1.37)*, Michael D. Tiemann, GNU C++ release.

Trademarks

The following are trademarks:

- Chorus – Chorus Systèmes
- DECstation – Digital Equipment Corporation
- microVAX-II – Digital Equipment Corporation

- Network File System – Sun Microsystems, Inc
- NFS – Sun Microsystems, Inc
- OSF/1 – Open Software Foundation
- SunOS – Sun Microsystems, Inc
- Ultrix – Digital Equipment Corporation
- UNIX - UNIX Systems Laboratories, Inc
- VAX – Digital Equipment Corporation

Chapter 1

Background to the Amadeus Project

The overall objective of the Amadeus Project is to design and develop an integrated application support environment for writing and running distributed applications which can manipulate persistent (long-lived) data. This release of Amadeus (v1.0) is a step towards meeting this overall goal by providing support for C++ above Ultrix in a distributed and persistent environment. Support will be extended to other Unix variants and well known programming languages in future releases.

A goal of the Amadeus project is to extend each language as little as possible. This has two major benefits: programmers do not require extensive retraining to use the Amadeus environment and existing code can be re-used as much as possible.

Ideally, no language extensions would be necessary to write applications that would run in the Amadeus environment. In practice, however, interfacing a language to the Amadeus environment requires that additional information be provided at runtime. Typically, a pre-processor is provided for each supported language which generates the necessary supplementary information and then calls the native compiler. A further practical concern is that it is not always possible to support all the features of a given language in Amadeus' distributed and persistent environment.

Amadeus v1.0 supports C++ with extensions; this extended language is called C**. Rather than using a pre-processor, the changes to support C**have been integrated into a C++ compiler. A description of the C**extensions to C++ is available in the *C** Programmers' Guide*.

1.1 Introduction to Distributed Processing

A distributed processing system can in principle include all types of computers, from microcomputers to mainframes. The distribution of processing power should support the functions required at different remote or local locations. Some distributed processing systems are homogeneous in the sense that the same operating system or the same CPU

architecture are used throughout the system. The larger a distributed processing system is, however, the more likely it is to be heterogeneous - composed of different hardware configurations and different operating systems.

The components of a distributed system are separate: ideally, each component can be relocated, new components can be added, and existing components can be reallocated. A distributed processing system can be expanded in small increments, on an as-needed basis, allowing systems to be tailored to suit exact processing requirements. Incremental growth leads to reduced cost as well as risk containment; applications can span technology changes and proprietary product boundaries.

The Amadeus v1.0 is limited to be homogeneous: it has been developed and tested on DECstations (and a Dec6300) and microVAXes running Ultrix, and all application programming is in the C**language. Future releases will extend the range of supported host CPU architectures, host operating systems and available programming languages.

1.1.1 Writing Distributed Applications

Applications that run in a distributed environment must be composed of clearly separable modules. These modules communicate with each other by explicitly passing data between modules or by sharing access to files or databases. In some distributed systems, it is also possible for modules running on different computers to share a common virtual address space, using the technique of distributed shared memory. However, Amadeus v1.0 does not use the concept of distributed shared memory: modules running on different machines cannot share a virtual address space.

Partitioning an application into separate modules permits the distribution of processing among computers in a building, or a university campus, or even across the world. When communications between modules is explicit, mechanisms can be designed that work whether the modules are on the same system, on the same local network, or separated by a wide area network. Using such technologies as the remote procedure call (RPC), eliminates the need to make changes to application programs to allow them to execute in different environments.

Some software modules provide a well-defined service to other modules; these service providers are called servers and are used in many different applications. Requesting modules, known as clients, access services by issuing requests to server modules. Service actions are initiated when a client sends a request to a server and are completed when the client has received a response. The roles of client and server are relative to a particular service being performed; a server may itself act as a client of another server to accomplish part of the computation it was asked to perform. A typical distributed processing system might contain servers to process electronic mail, query data bases, access a remote file system, or perform a long computation on a fast CPU. In a distributed processing system, clients and servers cooperate to execute an application.

In Amadeus v1.0, no particular servers are supplied as a part of the release, other than the Amadeus server required to suppport Amadeus itself. However the programming and execution environment provided by Amadeus aids the construction of both servers and clients.

1.1.1.1 Remote Procedure Calls

Remote Procedure Calls (RPCs) are one of the widely accepted mechanisms for extending programming languages to enable the writing of distributed applications. Modular programming techniques advocate partitioning programs into a number of procedures, so that each procedure can be tested separately and then combined to form the program. Data is passed between the procedures using argument lists. A call to a remote procedure appears similar to a call to a local procedure in the program. The difference is that the two procedures are not linked together; in fact, they need not reside on the same node. Programmers typically specify particular RPCs in an Interface Definition Language (IDL), which is used by the RPC compilation system to generate stubs. The remote procedure call facility takes care of converting between data formats when different operating systems or programming languages are used. RPC stubs contain information about the location of remote objects. Most remote procedure call facilities can transparently locate the program being called through the use of a directory or naming service.

In Amadeus v1.0, the C**compilation system includes its own RPC stub generation facility. C**programmers need not give interface definitions in a separate language: rather they can use C**itself.

1.1.1.2 Directory Service

Machines communicate with each other over a network using addresses that resemble international telephone numbers. Machines have no trouble keeping track of long strings of numbers, but people prefer to use names that they can remember easily. A directory service is a key component of a distributed processing system providing a mapping between names and machine addresses. Given the name of an object, the directory service can provide the address, or in some instances, other attributes of the name.

Within the Esprit Comandos-2 project, a distributed directory service is being built, based on the CCITT X.500 series of recommendations. In Amadeus v1.0, however, no high level naming or directory service is provided as a part of the release. Such a service is not necessary to build distributed programs, although it may often be useful.

1.2 Persistence

In procedural programming languages, we normally think of data as having a well-defined lifetime. A variable declared within a block or procedure will usually persist only during the activation of that segment of code and will not be accessible later; at best, it will survive over multiple activations of that code segment until the program exits. From the programmer's perspective, the lifetime of an object created as part of a data structure is the duration for which the object is accessible. The lifetimes of most objects are bounded by the duration of a program run. External files are usually the only exception to this lifetime limitation: if a programmer wants an entity to survive a program run, it must be inserted into a file or a database management system, written to disk, and then later explicitly retrived (and rebuilt) for reuse.

Persistence is a term used to mean continuous; something is said to persist if the cause is removed and the effect remains. In programming environments, it is often necessary to manipulate many persistent objects, such as application data, source code, interface specifications, and compiled code. The advantage of providing persistence for programmers working in an object-oriented environment is that pointers to objects do not have to be rebuilt every time the objects are retrieved from storage.

In Amadeus v1.0, persistency is provided for specific C**classes identified by the programmer.

1.2.1 Orthogonal Persistence

Persistence of an object should be independent of its other attributes: this ability is known as "orthogonal persistence". Orthogonality is convenient for the programmer, who may not know in advance that a certain type of object will need to persist or that a certain function may be passed persistent arguments. It is also convenient for the implementation of the runtime system, since it allows objects to be treated uniformly.

The provision of persistence in C**programs is orthogonal to all other properties: code can be written so that it will work with the same interpretation independent of the persistence of the data which it manipulates. The persistence of an object is not statically defined; when an object is created it will not become persistent unless it becomes referenced by another persistent object. Persistence is a dynamic attribute which can be gained or lost transparently during execution; not all objects of the same class may necessarily persist.

The advantages of this approach to persistence are:

- Saving coding effort when transferring objects to and from files or a DBMS;
- Presenting a conceptually simple, single program view of the objects;
- Benefitting from the type safety of the host language (here, C++) when applied to persistent data.

1.2.2 Garbage Collection

Object systems typically provide automatic collection of garbage. Traditionally, this has been a problem in most C++ environments in particular, because of the difficulties in recognising memory references.

Amadeus v1.0 performs a degree of garbage collection automatically. Garbage collection within an Ultrix process is supported, but all normal activity is suspended while the mark and sweep collector runs.

1.2.3 Persistence and Naming

Every object must have a name, otherwise it cannot be identified or differentiated from other objects. The unique name identifies the object and also provides a means of locating

the object. In the distributed environment offered by Amadeus, objects can be located locally or remotely, and may be in main memory or on disk. In Amadeus, an object is accessed by invoking it, and if necessary bringing it into main memory at the local or remote node. The access method used is transparent to the programmer, and works whether the object is currently local or is remote, or whether it is currently on disk or in main memory.

Objects are named by object identifiers and mechanisms are provided to ensure that the references remain valid even if the designated objects migrate or persist and are later accessed again. Object identifiers are returned by C**'s `new` and can be subsequently communicated by direct assignment or by parameter and result transmission during function calls. Once in main memory, object identifiers are direct pointers (as is normal in C++), and the C**invocation mechanisms incur no overhead compared with the normal C++ mechanisms.

An essential property of a language with persistence is that persistent objects may be manipulated using the same expression syntax as for non-persistent objects. In order to execute such an expression, a binding between symbols in the program and objects in persistent storage must exist. In Amadeus, this binding is implicit.

1.3 Concurrency

In writing distributed applications, and particularly servers, it is often convenient if a number of lightweight threads can be used within a single virtual address space. Typically, there is a thread for each current request being handled by the server. Further although not a specific goal of the Amadeus project, clearly applications written to run on parallel machines, including multiprocessors, must exploit tasking.

Amadeus provides support for both lightweight and heavyweight processes: Amadeus v1.0 includes a lightweight process package implemented on top of (heavyweight) Ultrix processes.

In addition however, the Amadeus programmer is encouraged to design and implement his modules and classes without regard to how they may be integrated together for a particular (distributed) application – taking this approach aids re-use of code. This philosophy extends to the concurrency system: it is possible to use a lightweight process to execute various objects; and later to distribute those objects without changing the tasking code. That is, both "lightweight" and "heavyweight" processes in Amadeus span multiple nodes. To distinguish from the traditional viewpoint, an Amadeus *job* is a collection of Amadeus *activities*. At any particular node, the activities of a specific job can share a common virtual address space with one another. However a job – and hence the activities within it – can span a number of such virtual address spaces.

Both jobs and activities are objects, in the sense that they provide various operations (such as `suspend`) which can be invoked independently of the current location of the job or activity within the distributed system. Likewise both categories of objects can be persistent: their (persistent) value is essentially the "outcome" of their execution.

1.3.1 Synchronisation

In a multitasking environment, synchronisation is clearly important. While there are many known synchronisation mechanisms, the relationship of synchronisation to inheritance in an object oriented system remains a keen research issue.

The Amadeus environment itself provides a very low level synchronisation mechanism, based on classical semaphores. However it is believed that more complex synchronisation mechanisms may be built for specific supported language environments, in accordance with the philosophy of each language. In Amadeus v1.0, C** does not provide any language specific techniques because C++ itself does not. Instead it is believed that certain standard C++ libraries providing features for concurrency and synchronisation can be adapted to run in C**: however this has not yet been attempted by us.

Chapter 2

The Amadeus Project

The Amadeus Project provides an integrated platform for the development of applications which run in a distributed and persistent environment. An application which runs in this environment is structured as a collection of co-operating objects which may be dispersed throughout a distributed UNIX system. This environment is called the Amadeus environment and provides the necessary low-level support for distributed and persistent programming, including:

- Transparent access to local and remote objects;
- Garbage collection;
- Dynamic link loading;
- Object sharing;
- Object storage and retrieval;
- Object location services.

The *Amadeus environment* consists of two components: the *generic runtime*, and the *Amadeus kernel*. The generic runtime is layered above the Amadeus kernel, and provides a language-independent method of accessing remote objects. The normal UNIX services are, in principle, still available in the Amadeus environment. The Amadeus environment is intended to be language independent, so that both existing and new languages can be used to write distributed and/or persistent applications.

At each Ultrix node at which Amadeus v1.0 runs, there must be an *Amadeus server* active. Each node can then have a varying number of application processes: each such Ultrix process is called an *Amadeus client* (of the server).

Future versions of Amadeus will provide associative access to objects; security mechanisms; atomic transactions; online management of applications, and replication.

2.1 Relationship to the Comandos Project

The Comandos project is being developed in two phases: Comandos-1, under the Esprit-I programme, and Comandos-2 under Esprit-II. The Comandos-1 project produced an early prototype of the Comandos model and Comandos-2 which is currently underway is now building an industrial prototype of the model. Figure 2.1 shows the Comandos Project including the planned host environments and language-specific runtimes. The application services planned for Comandos include:

- Administration services;
- Distributed directory service;
- Data management service;
- Type management service.

Language-specific runtimes are planned for:

- A new language, developed under Comandos, called Guide;
- C++;
- Eiffel.

2.1.1 Amadeus Implementation of the Comandos Model

The Amadeus Project is an implementation by TCD of selected components of the Comandos model (see figure 2.2).

2.2 Amadeus Terminology

This section defines the following terminology used in the Amadeus Project.

2.2.1 Objects

Amadeus manages *objects*. An Amadeus v1.0 object contains one or more C**objects. Amadeus does not know anything about the internal structure of the objects that it manipulates nor about the semantics implemented by those objects. It also assumes that each object is a fixed size. All objects are described by *classes* which define the state fields of the object and the operations available to manipulate this state. Classes are currently written in the C**programming language.

If Amadeus requires any language-specific information about an object, it makes an *upcall* to the appropriate language-specific runtime: in Amadeus v1.0, this language-specific runtime is additional code generated by the C**compiler. If Amadeus requires information

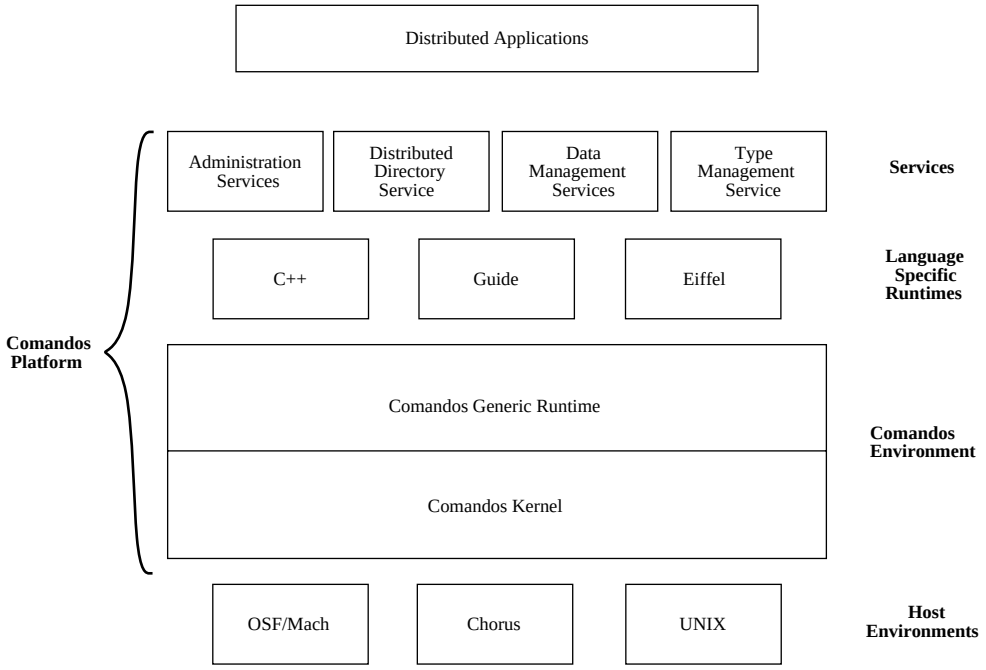

Figure 2.1: The Comandos Project

concerning an object in order to manipulate that object, this information must be provided by means of the upcall mechanism.

Amadeus can manipulate objects of any size, but each heap allocated object carries a fixed overhead for a header which is allocated for the object by the Amadeus generic runtime.

Most objects are passive (active objects which are called jobs and activities are discussed in Section 2.2.5 and Section 2.2.6) and are executed only as a result of an operation invocation on the object.

2.2.2 Context

A *context* is an abstraction of a virtual address space. It consists of a dynamically varying collection of objects co-located at the same node. In Amadeus v1.0, each context is implemented as a single Ultrix process.

2.2.3 Clusters

In Amadeus, the global space of objects is split into a set of *clusters*. Every persistent object is contained within exactly one cluster. The group of objects in a cluster may vary

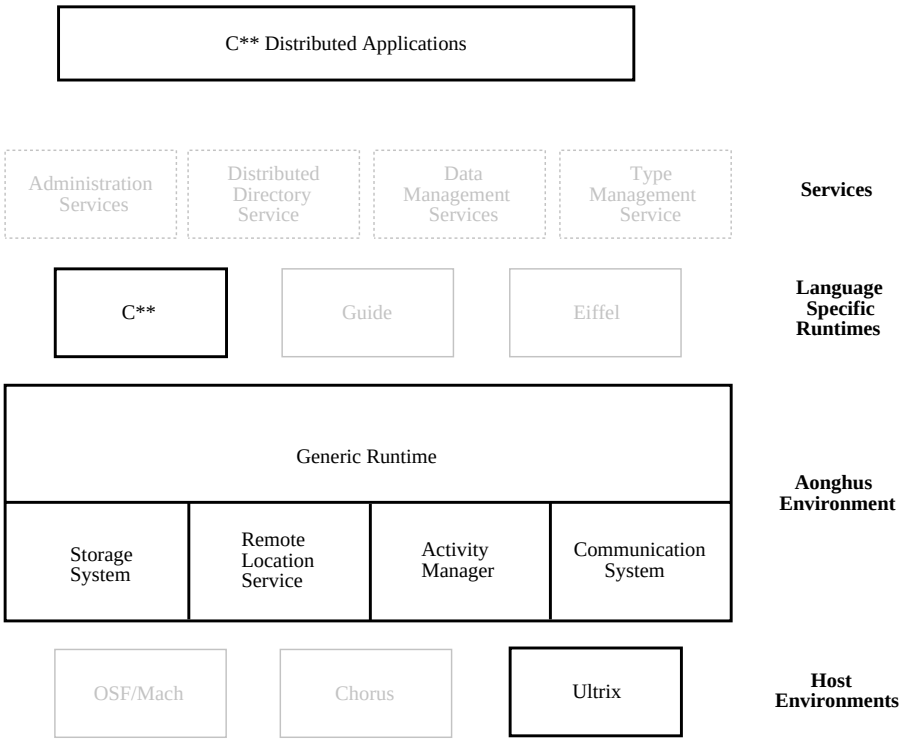

Figure 2.2: Amadeus v1.0 implementation of Comandos

dynamically either by creating new objects in the cluster; moving existing objects between clusters; or by garbage collection of objects in the cluster. Related objects can be grouped in the same cluster, so reducing the number of object faults when they are used together.

Clusters are the units of mapping into and unmapping from a context. Each context contains a set of clusters which may vary dynamically as more clusters are mapped into and out of the context. Currently clusters are usually only unmapped when the owning job terminates and there are no outstanding cross-context invocations active in the context. When an object is required by a job, the entire cluster containing the object is mapped into the appropriate context. Mapping a cluster is transparent to application code; there is no explicit primitive to fetch a cluster.

In Amadeus v1.0, each cluster is stored as a single Ultrix file.

2.2.4 Containers

Amadeus considers secondary storage to be divided into a set of *containers*. Each node may have zero, one, or more containers. Each container stores a subset of the clusters in the entire system. Amadeus does not support the migration of clusters between containers.

In Amadeus v1.0, each container is represented by an Ultrix directory. Further, it is

assumed that all the directories which support containers are available system wide via NFS.

2.2.5 Jobs

A *job* may be considered as a distributed (heavyweight) process. A job consists of a set of contexts (address spaces) and a set of *activities* (see section 2.2.6 below). At each node visited by a job, the job has a context into which clusters being used by the job at that node are mapped. Jobs may share objects, but they cannot share physical memory in Amadeus v1.0. A job is implemented as a collection of *Amadeus clients* – see section 2.2.8. There is one Amadeus client per node visited by the job, each running the same Ultrix text image.

Users can create new jobs by running Amadeus clients from the Ultrix shell. A new job can also be created from an application (the new job will have the same Ultrix text image as its creator). When a job is created, it consists of a single Ultrix process. Within this initial process, a single activity consisting of a single lightweight process is created. If the job was created from the shell, then this process executes the mainline of the Amadeus client application. If the job was created by another job then it carries out a specified operation on a specified object, as indicated in the invocation frame passed to it by its creator.

A job has a data structure within the data segment of the initial process which describes the current state of the job. Among the more important pieces of information maintained in this descriptor are a list of the nodes used by the job (used on job termination to release resources held by the job at these nodes) and a list of the ongoing activities of the job (used to detect the termination of the job).

New Amadeus clients are allocated to a job when it *diffuses*. A job diffuses whenever it first carries out any processing at a remote node. Such processing could be either a full invocation, or possibly as a result of internal housekeeping by Amadeus on behalf of the job. The client is created by the Amadeus server at the target node. Once created, the client idles until it (next) receives a remote request.

2.2.6 Activities

Activities are distributed lightweight threads of control, analagous to lightweight processes, but have the possibility of spanning different contexts. Normally, activities execute by synchronously invoking operations on objects.

An activity has a single supporting lightweight process when it is created. If the activity performs a remote or cross-context invocation, then this initial lightweight process in the initial context is blocked and a new lightweight process is created in the target context. If an activity performs a *rebounding call* back to an earlier context, then it will (conceptually at least) have two or more lightweight processes in a given context.

If the target object of an operation invocation is mapped into the same context as the invoking object, then the operation is carried out immediately. If a failure occurs, an

exception is returned. If the target object is not mapped into the current context, then an object fault is detected, which Amadeus will resolve. The Amadeus kernel must first locate the target object given the full system name for it. The object could be located in a context of another job at the current node, mapped at a remote node, or stored (locally or remotely) in the distributed storage system.

If mapped locally, an activity makes a cross-context invocation on an object. Otherwise the object fault may be resolved either by fetching the object from a remote node and mapping it into the activity's current context or by invoking the object remotely. Decisions about where the invocations are carried out are based on load-balancing criteria.

In Amadeus v1.0, a lightweight process package is provided for Ultrix. Future releases of Amadeus may use the thread support or lightweight process package (if any) of certain host operating systems.

2.2.7 Invocations

Invocation is, by default, location transparent: the invoker does not need to know the location (local or remote, on disk or in main memory) of the target object. It is also possible to override location transparency by specifying the node at which the invocation is to be carried out.

Although operation invocation is normally synchronous, it is also possible to invoke an object asynchronously by creating a new activity to carry out the required invocation.

2.2.8 Amadeus clients

Each Amadeus context is implemented as an Ultrix process known as an Amadeus *client*. The text of a client is created when an application is linked with the standard Amadeus library. A client is created when an application is run from the Ultrix shell or when a job diffuses to a new node during execution of an application. Each application runs as a new job. When that job diffuses, the same client text is used for the job at all nodes. However, running a client at nodes other than the initial node of the job, causes the client to idle waiting for a remote invocation request message rather than executing the application mainline. There may be many client images available in any Amadeus environment, and many jobs executing simultaneously, and running the same or different images.

2.2.9 Lightweight processes

Each Amadeus client includes a simple *lightweight process* package. Lightweight processes (or just "processes") are created in response to requests arriving on the network or from another client (cross-context invocation or job creation) or when a new activity is created in the current context.

Each process has a descriptor (including a pointer to the message that caused its creation and which is used for replies) and a stack. Currently, all stacks are allocated from the Ultrix heap and each is unprotected. Stacks are a fixed size so that stack overflow is possible and is undetected.

The lightweight scheduler selects lightweight processes based on their priority (which is inherited from the activity of which they are a part). Each process runs for a fixed time quantum unless it blocks voluntarily. Processes with an expired time quantum are pre-empted when crossing the generic runtime boundary (see section 3.2 below).

2.2.10 Stubs

A *stub* can be viewed as being a large object reference, holding information to not only find the object, but also to create a proxy (see section 2.2.11) for that object. At stub is only visible to the generic runtime (see Section 3.2). Although stubs which refer to the same target object could potentially be shared by objects in the same cluster, currently this is not supported.

2.2.11 Proxies

A *proxy* is the local representative of an absent object. Invocations of a proxy and of the associated true object are normally indistinguishable, except for the delays involved in accessing remote objects. A context has a single proxy for each remote object used by other objects in that context. A proxy is the same size as its true associated object, and may be replaced by the true object if the true object is brought into the proxy's context.

2.2.12 Exceptions

An *exception* is an unusual condition arising within the Amadeus environment itself, or within the underlying host operating system, or within application code. The generic runtime will propagate an exception, if necessary, which arose at a remote node; it will indicate exceptional conditions when appropriate to each language specific runtime.

In the case of C++, there is not yet a stable implementation of support for exceptions. Pending this, in C**the programmer can catch exceptional conditions in a manner similar to that of a failure of `new`, as explained on page 280 of the Annotated C++ Reference Manual.

2.3 Amadeus v1.0

The Amadeus v1.0 distribution consists of three components: the Amadeus server; the Amadeus library; the C**compiler and a utility which can be used to configure an Amadeus system dynamically. A single instance of the Amadeus server must run on every node in the system, and the Amadeus library must be linked to each application compiled using C**.

The server supports a remote invocation protocol, the Inter-Kernel Message (IKM). This protocol provides a mapping from job name to local Ultrix process name: the mapping is required when sending a message to the local part of the job for the first time from a remote node. The server also implements job diffusion and initialises global (remote)

nodes. The library contains all the remaining code necessary to support the Amadeus environment.

An overview of the kernel and library is given in the following chapter.

Chapter 3

Amadeus Environment

The Amadeus server and Amadeus library together form the Amadeus environment. Internally, the Amadeus environment consists of the following five components:

- Storage System providing long-term storage for clusters.
- Generic Runtime allowing applications written in conventional object-oriented languages to use distributed and persistent objects.
- Remote Location Service which locates clusters in a distributed system.
- Activity Manager which provides facilities to create jobs and activities and implements operations on these. This component also provides a remote and cross-context invocation service and some low-level synchronisation facilities.
- Communication System which implements the inter-kernel message (IKM) protocol supporting remote invocation.

3.1 The Storage System

In a quiescent Amadeus system, all objects are stored in the storage system. This consists of a collection of containers which are the basic units of storage distribution.

3.1.1 Container Implementation

In Amadeus v1.0, each container is implemented as an Ultrix directory with each cluster stored in the container stored as a file in that directory. Each container is stored entirely at one node, although many containers may be stored at the same node.

Every node participating in an Amadeus environment must use NFS to import *all* the directories implementing containers. On each node, all the containers should appear with the same path name in the local directory hierarchy. The storage system always reads/writes files as if they were stored locally, relying on NFS to implement remote fetch and store.

3.1.2 Naming

Within Amadeus v1.0, containers are known by 8-bit identifiers. The storage system is also responsible for allocating cluster identifiers, and per container object identifiers, both of which are 24-bit identifiers. There is a maximum of $2^8 * 2^{24}$ globally known objects in Amadeus v1.0, partitioned in up to 2^{24} objects in each of 2^8 containers. Each container holds its objects in up to 2^{24} clusters. A stub contains the global name for an object, together with a hint as to in which cluster the object may be found.

3.1.3 Storage System Interface

The current storage system provides a simple interface which has the ability to retrieve or store a whole cluster at a time. The storage system also provides a routine to allocate a new cluster identifier in a specified container. Clusters are created in virtual memory and can be written to the storage system only after they have been allocated an identifier.

The storage system also provides a primitive to allocate an object identifier in a specified container.

3.1.4 Object and Cluster Storage

A cluster consists of a header and a body containing the objects currently stored within the cluster.

Each object is stored contiguously within the cluster body. An object itself consists of a header followed immediately by the object's state, which is in turn followed by a set of stubs for referenced objects.

3.2 The Generic Runtime

The generic runtime enables applications written in a conventional object-oriented language to use distributed and persistent objects. Rather than putting most of the burden for doing this on the compiler, the generic runtime performs many of the actions required in a language independent manner. Whenever language specific information or actions are required, the generic runtime makes an upcall to code supplied by the language (or application).

Currently the main responsibilities of the generic runtime include:

- Detecting object/cluster faults;
- Mapping and unmapping clusters;
- Marshalling parameters for remote invocations;
- Dispatching incoming invocations;
- Managing object creation and promotion;

- Garbage collection.

3.2.1 Proxies and object faults

When an object is mapped into some context it uses the normal object reference format of its host programming language. Specifically C**uses direct memory references, like C++. However, since some of the referenced objects may not be present in the current context, an object reference may refer to a proxy rather than the true object. A proxy is the local representative of an absent object. When the referenced object is invoked, the code associated with the proxy generates a fault on the object's cluster.

When an object fault is detected, the generic runtime is called.

The generic runtime in turn calls the Remote Location Service (see Section 3.3) passing it the hint for the object's cluster, in order to locate the cluster and make it available for mapping by the generic runtime into the current context. This may not always be possible since each cluster can only be mapped once anywhere in the system and may already be mapped elsewhere. The Remote Location Service either returns an indication back to the generic runtime telling it to map the cluster locally or perform a remote operation.

If the invocation could be remote, then the generic runtime builds a parameter frame describing the invocation allocating space and upcalling to the proxy to marshal the parameters. The generic runtime then calls the Remote Location Service with the name of the cluster and a pointer to the parameter frame. The Remote Location Service locates the cluster and either forwards the invocation to it or makes the cluster available to be mapped locally. The Remote Location Service either overwrites the original parameter frame with the result block or returns an indication to the generic runtime to map the cluster.

3.2.2 Mapping and unmapping objects

A context is made up of the following components:

- The set of clusters currently mapped into it, and an associated table of their position;
- A registry of all the objects currently mapped into the context;
- A heap holding newly created objects and newly created proxies;
- A stack per lightweight process;
- Executable code.

When mapped, each cluster is normally registered in a (per context) table maintained by the generic runtime. Mapped but unregistered clusters may exist, but this is a transitory state during unmapping. When mapped into a context, a cluster is stored contiguously in the context, except that some of the objects may be copied out onto the heap when the cluster is mapped in (space is still reserved for them in the cluster).

When a cluster is mapped into a context, all objects in the cluster are registered in the object registry and any objects for which a proxy does not already exist in the context enter the *registered* state. There is one object registry per context. Again, no code is directly linked with registered objects.

When an object is invoked for the first time, there are overheads while it is being bound to its class code. After the first invocation, the cost is much cheaper, and equal to that incurred in C++.

3.3 The Remote Location Service

If a cluster fault occurs, it is necessary to locate the target object given its generic identifier. The Remote Location Service is responsible for cluster location.

3.3.1 Data Structures

The Remote Location Service maintains location information for clusters in a number of tables. Each node has a Kernel Cluster Table (KCT) which is shared by all Amadeus clients and the local Amadeus server and which contains an entry for every cluster mapped at the node providing the identity of the context into which it is mapped. Every node at which a container is mounted has a Mapped Cluster Table (MCT) for each container mounted at that node; each node has one MCT.

The MCT contains one entry for each cluster from the corresponding container that is currently mapped at some node (except those mapped at the control node itself), giving the identity of the node at which the cluster is mapped. The Remote Location Service also maintains a cache of cluster-to-node mappings at every node. This cache contains entries for clusters that are neither stored nor mapped locally, but have been invoked recently from the current node.

Each of these tables is stored in Ultrix shared memory and accessible by all clients and the server at each node.

3.4 The Activity Manager

The Activity Manager manages jobs and activities including implementing remote and cross-context invocation. A lightweight threads package implements activities.

3.5 The Communications System

The Communications System allows communication between nodes and between contexts on the same node which is the basis for remote and cross-context invocations and Remote Location Service requests. The Communications System is built on top of the underlying Ultrix socket interface and uses UDP internet domain sockets for remote communication and UNIX domain sockets for local communication.

While such a protocol would not normally be used for local communications, use of the Inter-Kernel Message (IKM) protocol over UNIX domain sockets overcomes the limitations on UNIX domain message sizes. This makes the difference between local and remote communication transparent to the rest of the Communications System and at higher levels.

In principle, the IKM protocol of Amadeus could be replaced by another RPC like protocol[1].

3.5.1 Implementation of IKM

Each Amadeus client has two sockets associated with it on which it recieves IKM messages:

- An internet domain socket on which it recieves remote requests;
- A UNIX domain socket on which it recieves cross-context invocation requests.

When sending a remote request, a job needs to obtain the port number for its client on the target node, which is normally cached locally. If, however, the job has not previously carried out an invocation from the current node to this target node the port will not be available locally.

To find the port, the Amadeus server (which must run on each node) is used. The main function of the server is to maintain the mapping between job identifiers and the local port for that job, for all jobs present on that node. A data structure known as the *Address Translation Table (ATT)* is used, which maps a *jobid* and *nodeid* to the port to use to talk to the job at that node.

When a job needs to know its port on some node it can query the server at that node by means of the *GetPort* request. The address of the server on each node is well known.

Within each client, incoming messages are recieved by a `SIGIO` handler associated with the two sockets. New requests are dispatched by the IKM by creating a new process to handle the request.

[1] Although the IKM implementation is reasonably fast: about 8msecs round-trip for a distributed C**program on DS2100s