Technical Report

UCAM-CL-TR-73
ISSN 1476-2986

Number 73

Resource management in a distributed computing system

Daniel Hammond Craft

This technical report is based on a dissertation submitted March 1985 by the author for the degree of Doctor of Philosophy to the University of Cambridge, St John's College.

ISSN 1476-2986

Resource Management in a Distributed Computing System

Daniel H. Craft
University of Cambridge
Computer Laboratory

Abstract:

The Cambridge Distributed System, based on the Cambridge Ring local area network, includes a heterogeneous collection of machines known as the *processor bank*. These machines may run network servers or may be loaded with services and allocated to users dynamically. The machines and the variety of services they can support (e.g. different operating systems, compilers, formatters) are viewed as resources available to other components of the distributed system.

By using a processor bank, two fundamental limitations of the personal computer approach to distributed computing can be overcome: responsiveness for computation-intensive tasks is not limited by the single, personal machine because tasks may expand into processor bank machines as necessary; and applications are not limited to the operating system or languages available on the personal computer because all of the systems and languages which run on processor bank machines are at the user's disposal, both for implementing new applications and importing programs from other systems. Resource management is seen as one of four areas which must be addressed to realize these advantages.

The resource management system must match client requirements for resources to those resources which are available on the network. To do this, it maintains two data bases: one contains information describing existing resources, and the other contains information indicating how to obtain resources from servers or have them constructed from existing subresources by fabricators. The resource management system accepts resource requirements from clients and picks from the alternatives in these data bases the "best" match (as defined by the resource management policy).

The resource management issues addressed include: resource description, location and allocation, construction, monitoring and reclamation, authentication and protection, and policy. The design and implementation of two resource management servers is discussed.

Acknowledgements

I should like to thank my supervisors, Professor David Wheeler (who supervised me in my first year) and Dr. Andrew Herbert for their guidance. I have only praise for Andrew as a supervisor - he is genuinely interested in his students, becoming involved and offering ideas without interfering. The erudite Professor Roger Needham has also contributed much to my education at Cambridge. As head of department, Roger's talent for combining laissez-faire and subtle intervention has brought the Laboratory much of its success.

I should like to thank Andrew Herbert, Mike Schroeder, and Jim Mitchell who were kind enough to read and criticize my thesis. Their suggestions were invaluable in improving its organization and presentation. I owe them much.

I am grateful to the members of Project Mayflower, particularly Graham Hamilton and Andrew Herbert with whom I shared the early days of the "angry young men" - there seemed so much to do and so little time in which to do it. Mayflower has provided an excellent forum for presenting and developing ideas, as well as a small but demanding user community. I am also grateful to the members of the old Systems Research Group whose ideas and labor produced the Cambridge Distributed System and the philosophies behind it. I should particularly like to thank Jon Gibbons, Nick Ody, and Gray Girling who pointed out requirements and provided feedback when I was first collecting my ideas on resource management.

As a Marshall Scholar, I have kindly been afforded the opportunity to study at Cambridge, and to experience British and other European cultures. For this I am eternally indebted to the Marshall Aid Commission and to the British people. I am also indebted to the Lundgren Foundation for their financial assistance.

I should like to thank Bev, Jon, Graham, Steve, Nick, and Harry for their friendship and contributions to my thesis through discussion and hortative support.

I am deeply grateful to my family for their love and support. My parents' unceasing emphasis on education and assiduousness has provided me with many opportunities.

Except where otherwise stated in the text, this dissertation is my own work, and includes nothing which is the outcome of work done in collaboration.

List of Chapters

Table of Contents

1. Introduction

1.1. Thesis

In a centralized computer system, there is a single operating system which understands and controls the available resources such as processor cycles, memory, files, and attached peripherals. The basic resources of concern in a traditional operating system are of one of a small number of types, each of which is managed by a very specialized program. These programs (e.g. schedulers, virtual memory systems, filing systems) have been highly tailored to provide rapid allocation and good utilization of the resources. To achieve acceptable performance, they must often be included in the system kernel.

Commonality

Resources in a distributed system are varied and prolific. Hence there is an overriding desire to have a single framework for their management. Many of the mechansims for managing different types of resources are common and thus should not have to be duplicated. Additionally, having all of the information about available resources conveniently available in a consistent form makes it possible to have a policy incorporating multiple types and instances of resources.

The thesis of this dissertation is that there is sufficient commonality in the requirements for the management of resources that they can effectively be collected into a resource management system. This minimizes the requirement for resource management mechanisms to be provided individually in external components, thus easing the introduction of new types of resources into the system. It also enforces a consistent interface for offering, requesting, and conversing about resources, thus reducing the number of interfaces a program must understand to manipulate different resources. The cost of such generality is loss of the ability to tailor highly the management of resources with the result that they are less efficiently utilized. It is claimed that such cost is far outweighed by the advantages.

Overview

The issues of resource management include:

- *description of resources* - so that clients may offer, request, or otherwise converse about a wide variety of resources in a consistent manner;
- *location and allocation* - to match requests for resources with those which are available on the network;
- *construction* - the higher level resources in which users are generally interested must be provided by combining or augmenting the raw, lower level ones;
- *monitoring and reclamation* - to detect resources inaccessible as a result of a crash so that they may be reclaimed or appropriate parties may be notified for cleanup or recovery;
- *authentication and protection* - clients requesting resources must be ensured that the resources received are authentic, while resources (or their providers) must be ensured they are allocated only to clients privileged to use them; and
- *management policy* - policy guides the decision among alternatives for allocation, construction, and preemption, as well as whether requests should be granted at all.

Also of interest are:

- *considerations of optimization* - including reservations of resource sub-components before construction, and maintenance of a pool of spare resources likely to be requested;
- *instantiation of the resource management system* - the distributed system must be bootstrapped through the level of the resource management system before its facilities can be used;
- *related work or alternatives* - different approaches to resource management; and
- *integration into the distributed system* - different uses of the resource management system.

All of these topics are discussed in this dissertation, in approximately the above order.

1.2. Background

The setting for this research is the Cambridge Distributed Computing System [Needham 82] based on the Cambridge Ring local area network [Wilkes 79] (see figure 1.1). The system includes a few dedicated machines running network services and several traditional timesharing systems, but its main computing power is provided by the heterogeneous collection of machines known as the *processor bank*. These machines may run network services or they may be allocated to run user applications. All these functions, and the machines on which they run are viewed as system-wide resources for which management must be provided.

Processor Bank Systems

The processor bank machines are commonly used to run one of two local operating systems, Tripos or Mayflower. They are also used to run network services, such as the Tripos Filing Machine mentioned below.

Tripos is a portable operating system for minicomputers written in and thoroughly oriented towards the language BCPL. It is a multi-tasking, message based, single user system. Tripos is available on processor bank Computer Automation LSI4s and Motorola 68000s. As these have no local discs, Tripos keeps its files on the network fileservers, and as they have no directly attached terminals, Tripos accesses terminals through a terminal concentrator server. It has been the primary operating system for systems research in the Laboratory. [Richards 79, Knight 82]

The *Tripos Filing Machine* server imposes a Tripos filing system structure on the objects provided by the network fileservers. It accepts Tripos filing system commands from clients, and specifies to the fileservers operations on fileserver files and indices. It provides caching and read ahead, thereby reducing the overall load on the fileservers and improving response time to clients. Also, the filing system code in the Tripos operating system is reduced to a stub which interacts with the filing machine. [Richardson 84, TriposFM 83]

The *Mayflower operating system* is a multi-tasking, monitor based operating system which runs on the processor bank 68000s. It is a preferred

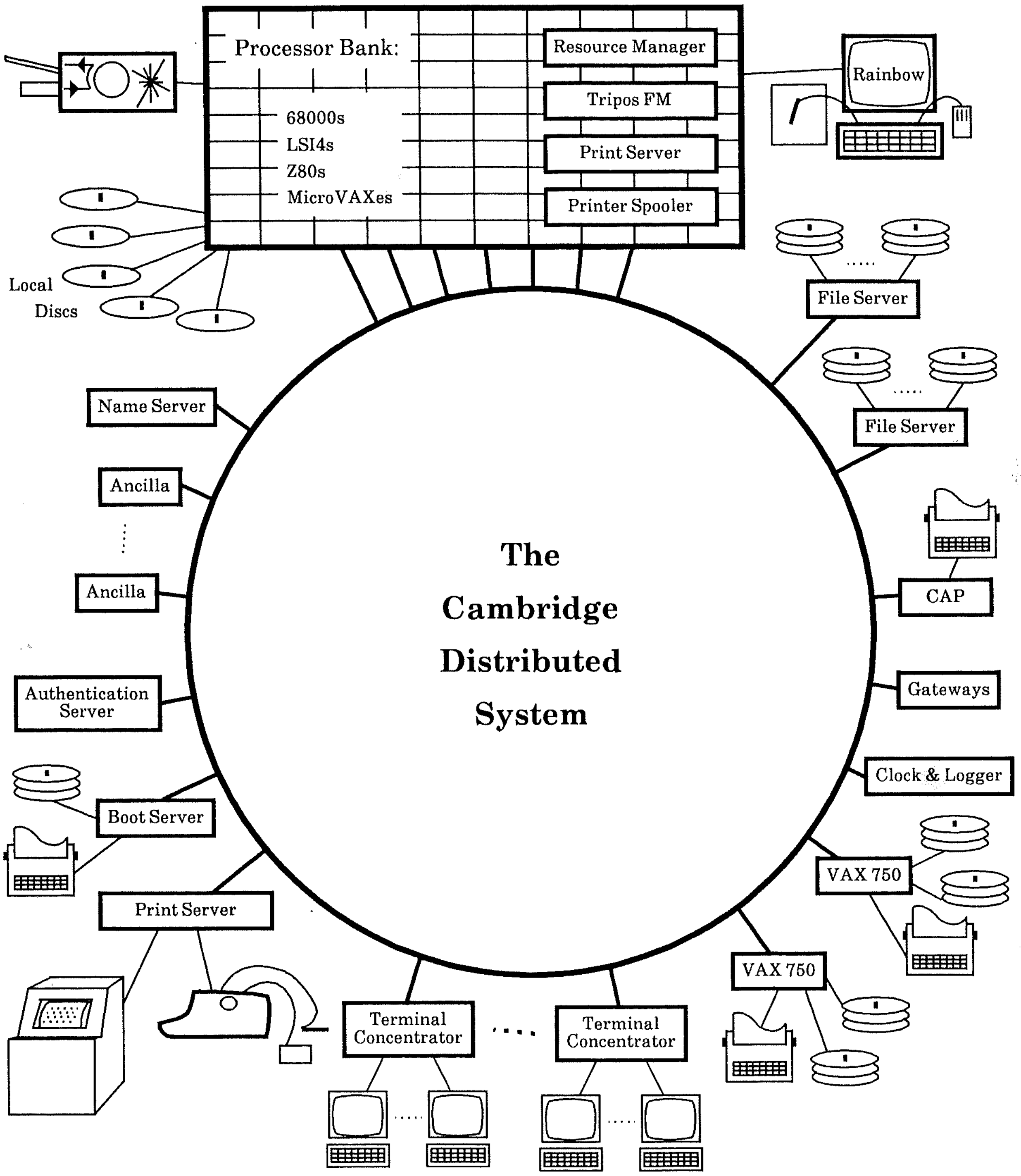


Figure 1.1. The Cambridge Distributed System

language system: the kernel interfaces are at the bit and byte level, but most of the library support is based on the CLU language [Liskov81] which has been extended with concurrency and synchronization primitives, as well as with remote procedure calls for cross machine communication. It is part of Project Mayflower, which is developing an environment to facilitate distributed programming. [Mayflower84, CLU RPC84]

There are a number of other systems available for the processor bank machines. These include: a restricted version of Tripos for undergraduates, a mail reader/composer and daemon for the Ring mail system, a garbage collector for the File Servers, a semi-automatic Wirewrap system, an account daemon for the Tripos Filing Machine, and a Tripos Mail Server.

Dedicated Network Servers

Services are run on dedicated machines if there is some expensive or special purpose hardware necessary, as with a printserver or a fileserver, or if there are special requirements for location or initialization, as with a nameserver or an authentication server. Some of these machines may be included in the processor bank, noting their special hardware or importance, so that the services they run may be instantiated and monitored using the standard processor bank mechanisms.

The Cambridge *Name Server* translates a textual service or machine name into an absolute Ring address. In a distributed system, services may be moved from one machine to another and machines may be moved to new positions on the network. The Name Server provides an indirection so that service or machine locations may be changed without having to change the code of their clients.

The Cambridge "universal" *File Servers* provide files and indices (lists of file pointers and other indices) on which clients can impose their own filing systems. Atomic file updates are provided for "special" files. There are currently filing systems for Tripos, the Cambridge CAP computer (running a capability-based timesharing system), Mayflower, and several of the small servers. [Birrell80, Dion81]

The *Active Object Table (AOT)* service provides mechanisms for authentication and protection. It issues *uidsets* for people or services when they become active in the distributed system. For example, a uidset will be created when a user's password is presented to the AOT by the service to whom he typed it at login, or when a service is loaded. A uidset contains 64 bit unique identifiers for the object's name, its type (e.g. *user* or *service*), an access key, and a control key. If appropriate parts of the uidset are included in a request to a service, the service can authentically determine the identity of the client (or the client on whose behalf he is acting) as well as what privileges the client possesses. [Girling 82]

The *Ancillae* are loading services used to hide the details of loading processors. There is one for each type of processor in the processor bank. An Ancilla will accept a request to load a file from the Ancilla filing system (kept on the File Servers) into a named machine. It will also accept a request to reset a named machine (equivalent to pushing the "reset" button).

There are several *Print Servers* which control the various printers, including one for a laser printer which accepts the intermediate codes of several formatters as well as simple character streams or bitmaps. Spooling for the Print Servers is provided by the *Spooler*, a small server which buffers client documents on the File Servers until the appropriate printer is available.

The *Boot Server* is responsible for bootstrapping the distributed system after shutdown, and for reloading low level services if they crash.

Terminals and Workstations

Terminals are connected to the Ring through *Terminal Concentrator* servers, up to eight per concentrator. A user may have connections from his terminal to multiple machines, from which he may select one for interaction at a time. A connection to a service may be opened by specifying the name of the desired service to the concentrator monitor program. No screen management is provided for interaction via the different connections. [Ody 84]

Preferably, the user should be able to view the system from a small personal workstation on his desk. This home machine would manage a screen and network connections, and provide an interface to the rest of the world. Through this interface the user could access the resources of network services as well as the raw computing cycles provided by the processor bank machines. Xerox Dandelion workstations running the Xerox Development Environment [XDE 84] are currently being integrated into the system.

1.3. The Processor Bank Philosophy

Organization

Computers of various types and sizes are collected into a pool. Most of these will be available to support services on demand, though some may be reserved for dedicated servers. User requirements for a service may be accommodated by loading the service dynamically on one or more processor bank machines allocated for the request. Alternatively, the required service may already be loaded on a processor bank machine and can be allocated directly. In the Cambridge distributed system the processor bank does not incorporate all of the available computing power, so user requirements may also be satisfied by services external to the processor bank.

The processor bank is viewed as consisting of a number of small to medium scale computers, such as Motorola 68000s and Digital Equipment Corporation Vaxes, a few large mainframes, and several special purpose machines. This allows all of the users access to a very large amount of computing power which may be shared between them. The prototype Cambridge processor bank is necessarily somewhat scaled down. It includes Z80s, LSI4s, and 68000s, with MicroVAXes currently being integrated.

Alternative Philosophies

The processor bank philosophy contrasts sharply with that of *dedicated personal computers*. With the latter, each user is provided with a powerful personal machine on which he does almost all of his work. This computer is connected to a high speed local area network over which it communicates with other computers and servers, much in the same way as the user's small personal workstation in the processor bank scenario. The two approaches differ in where most of the work is carried out: in the user's personal computer or in processor bank machines.

With the personal computer approach, the display and discs are attached to the processor performing the user's task, thus allowing high bandwidth communication among these components. Their separation, which occurs with the processor bank approach, may result in decreased performance depending on task distribution and interfaces. (Requirements for avoiding this are discussed in the "Research Topics" section below.) The personal computer approach also has the advantage that the user is guaranteed availability of his machine.

Traditional *timesharing systems* enjoy the benefits of file and environment sharing. Retaining these advantages after distribution is difficult, but it has been done (e.g. in Locus [Popek81]). The primary problem with timesharing systems is that they suffer degradation under load, with the accompanying unpredictable response times. The processor bank approach has a number of potential advantages over both the personal computer and the timesharing system approaches, some harder to realize than others.

Potential Advantages

Users' requirements for computing power vary considerably. The processor bank approach allows the requirements of the task being performed to be matched to the attributes (e.g. speed, size, specialized hardware) of the machine used. It is not necessary to provide each user with a machine capable of handling the largest task that user will execute. Instead, users can be given a modestly powerful machine capable of managing a screen and coordinating interaction with user tasks, and major work may be executed by an appropriate machine selected from the processor bank.

Because the processor bank contains different types of machines it may support a wide variety of operating systems, languages, and tools. This facilitates importing existing software. Witness to the importance of this capability is the popularity of Unix, much of which derives from its plethora of available applications. It also allows the programmer to choose the most appropriate language and system for implementing new applications.

The processor bank facilitates the acquisition of multiple machines for a single user. Thus if a user wishes to accomplish several tasks concurrently or a single task which can be executed across multiple machines, he may experience a reduction in the real time required to accomplish them. Rather than multiplexing all of the user's tasks on one machine, they may be farmed out to multiple processors.

Because the computing power of the system is allocated on a per request basis, the apparent aggregate power of the system is greater than if machines were statically allocated to users. The alternate perspective is that fewer machines are required to provide the same degree of service to users, though any guarantee about machine availability becomes probabilistic with high dependence on the allocation policy.

The processor bank consists of racks of components: processors, memory, and network interfaces connected by back planes. Some of the machines have local discs. Others have no directly connected peripherals other than the network, and rely on network servers (which can utilize expensive peripherals more cost effectively) to satisfy peripheral requirements. By eliminating the need for peripherals, casing, and cooling fans for each machine, the cost of the individual computers is greatly reduced. By collecting the computers together, cooling and humidity control as well as power requirements are centralized and thus more easily handled. By placing the machines in a room separate from users, the problem of noise abatement is removed. (Some of these advantages may be accrued with personal computers if the processors and discs may be placed in, for example, the basement, leaving only the displays in users' offices.)

Research Topics

There are a number of research topics which must be considered before all of the above advantages of the processor bank philosophy can be realized. This dissertation addresses only the first of those listed below. The remainder are either under exploration or as yet unexplored at Cambridge.

There must be some management of the machines in the processor bank and the services which they can support. Though the processor bank provides the potential for matching tasks to appropriate available machines, some service must actually perform the matching. Though the processor bank allows multiple machines to be allocated to a user, some service must actually coordinate allocations. As users will be more interested in the services which processor bank machines can support than in the raw machines, some service must construct these higher level resources. These requirements are addressed in this dissertation.

Remote access to the user's screen and input devices must be provided so that programs running in processor bank machines may interact with the user in much the same way as a program running in a user's personal computer could. The difficulty is that the network is in between, thus reducing the bandwidth. The virtual screen interface must be chosen to short circuit the critical functions (e.g. mouse cursor update, menu handling) in the workstation, yet to notify the program of requests made by the user which affect that program's display (e.g. change of window size). The protocol must be language and system independent: cross language, type checked remote procedure call (RPC) is a preferred option.

There must be a program in the user's workstation that coordinates his tasks running on different machines in the network. It must translate requirements expressed by the user into requests for processor bank resources. These resources must be instructed as to their purpose, data must be routed between them as appropriate, and they must be monitored for failure or errors.

A user's files must reside primarily on network fileservers, as there is no single machine on which most programs accessing the files will run. These files must be readily accessible to the processor bank machines as well as to the user's workstation. The network fileservers and the systems running on both the processor bank machines and the workstation must support this approach.

2. An Initial Solution

The initial need for resource management came with the arrival of the processor bank. A Resource Manager server was built to satisfy the requirements as perceived at that time. This chapter discusses the initial system and its evolution, and then gives a historical view of the problems and further requirements.

2.1. The First Resource Manager

In the summer of 1980 Cambridge had six LSI4s in the processor bank. We ran the Tripos operating system, which was originally designed for machines with directly connected terminals and discs. The terminal driver and filing system had been modified so that these resources could be accessed across the network. The Tripos file handler imposed a filing system structure on files and indices provided in the Cambridge File Server. The virtual terminal handler would accept Cambridge virtual terminal protocol (VTP) opens on a particular port.

Terminals were connected to the Ring through a terminal concentrator. A user could direct the terminal concentrator to open a terminal stream to one of the Triposes by stating the machine name of one of the machines thought to be running Tripos. At this point, the terminal concentrator would retrieve the station number associated with the specified machine from the nameserver, and then make a connection to that machine. If the machine was running Tripos and there was no one logged in, then the connection would be accepted and the user would be asked to log in. Otherwise, the connection would fail and the user would try another machine.

The Tripos operating systems running in the machines crashed frequently in the early days. (Tripos has no memory protection, and it and most of its applications are written in BCPL, a non-typed language.) If a user noticed that a Tripos had crashed he would set about rebooting it. This involved pushing the reset button on the dead machine, obtaining a running Tripos, and instructing it to load a small bootstrap program into the dead machine. This bootstrap program would then read the full Tripos system from the fileserver. If there was no Tripos system running, then the single machine which had floppy discs was used; the bootstrap program was read off of floppy disc via a program in PROM. Detecting a crashed system was not difficult for the user to whom the system was allocated - his terminal connection died. However, a system which crashed when not allocated or during allocation, or which was left by the user on whom it crashed was only detected if a user became suspicious when he could not obtain a system. He determined if any of the machines were dead by asking all the other users which machines they were currently using. This was practical only because all of the terminals were in one room.

There was clearly a need for some management of these systems, so that users would not have to successively try the known machines until an available one was found, so that machines could be reloaded for each user request as Tripos is an unprotected operating system, and so that crashed systems could be detected.

Andrew Herbert wrote a Resource Manager that knew about the six machines and how to load programs into them.[1] It would search a table of machine states until it found an available machine which could run the requested system (only Tripos was available at that time), ask the Ancilla to load this machine, and then allocate it to the user. This Resource Manager (RM I) would accept VTP open requests from the terminal concentrator and then hold the open VTP stream in abeyance while the system was loaded. The loaded system would then open a second stream into Resource Manager, at which point RM would replug[2] the two streams.

RM I Evolved

The processor bank and the Cambridge Distributed System grew, and Resource Manager grew with them. Replug was replaced by reverse

[1]The author took over the responsibility for RM that October (1980).

[2]*Replug* was a facility of the byte stream protocol at that time which allowed an intervening party to notify the remote ends of two streams that the streams were being reset, and that further communication should take place on the given station and port.

connection, a second type of processor was added to the processor bank, and the one Ring was replaced by two Rings linked by a bridge. The initial six LSI4s in the processor bank became 38 machines including LSI4s, 68000s, the Rainbow high resolution display, and a Canon laser printer. The single configuration of Tripos became 12 configurations of 8 different systems.

RM I was written in Algol68C, initially to run in a Z80 with 32K bytes of RAM. Because of the lack of code space and the increased size of internal tables as the size of the processor bank grew, Resource Manager was functionally split into a Session Manager (which understood VTP streams and could provide a terminal interface for making enquiries and specifying particular systems) and the Resource Manager proper (which understood the machines and systems available, and could contact the Ancilla to have them loaded). RM continued to evolve over the next couple of years, and a number of experimental modifications were made, some of which were discarded. It is described below in its final form.

2.2. Facilities Provided

When RM is initialized, it has a number of known machines to manage. Each is endowed with ***attributes*** (from a fixed set of 16) which define the properties of the machine, such as its processor type, how much memory it has, and what peripherals are attached. There is a fixed allocation order of machines intended to cherish those with more valuable attributes. Each machine has an associated Ancilla which can be instructed to load code into the machine.

RM has knowledge of several different systems which it can load into machines to be allocated to the client. Each system consists of one or more load configurations matching Ancilla filenames with machine attributes required to run them, thus allowing a given system to be available on different types of machines. The client may also describe his own system by giving an Ancilla filename to be loaded and the machine attributes required to run it. A client request includes a system name or description, the desired machine attributes, and an allocation time for which the machine is required.

Allocated machines are monitored using a ***dead man's handle*** mechanism. For this mechanism, each allocated machine has an associated refresh time (on the order of minutes) within which the loaded system must contact RM with a request that this time be extended. While this mechanism can be fooled by software which crashes in part but continues to refresh, such machines will eventually be reclaimed when the overall allocation time elapses.

There are typically 350 to 400 requests to RM for systems per weekday. The majority of these are from users wishing to use the mail system or one of the operating systems available on the processor bank machines. RM's clients also include the File Servers requesting a Garbage Collector, the Mail System requesting a Mail Daemon, and the Pointing Machine controller requesting a Wirewrap system, among others.

2.3. Additional Facilities Desired

The first Resource Manager was designed to support the original Cambridge Model Distributed System [CMDS 80] in which machines from the processor bank were allocated as personal computers for the duration of a login session. Except for the lack of preloading to reduce user wait time, it met this aim well. However, as the potential of the processor bank philosophy was more fully exploited and as there was a need to incorporate resources outside of the processor bank, the limitations of the design goals became apparent.

Multiple Level Resources

RM I had a simple view of resources. It could manage a fixed number of machines, loading a memory image into them before allocation. This proved a limitation on several occasions. There was no concept of a resource as a multiple level object, so a command running on an operating system could only be accommodated as a resource by linking the command with the o.s. image. Each system had to be stored as a fully linked image, thus losing the advantages of delayed binding:

- considerable space was required to store all network systems;
- the images had to be relinked if there was a new version of any included code, e.g. libraries;

- there could be no sharing of code;
- the elegance of abstraction which late binding provides was lost (i.e. abstract interfaces which were provided with a new and faster implementation would not automatically be incorporated into the prelinked system); and
- no common parts of systems (e.g. operating system kernels) could be preloaded without compromising the choice for the final level of the resource.

We experimented with a three level resource for file transfer. A special "support" version of the Tripos operating system was loaded via RM and then itself determined the particular command to execute based upon the function code for the requested resource. This function code was originally obtained from the nameserver and was passed to the operating system by RM. The approach was inelegant as it depended on the nameserver having an entry for each such service which had to be coordinated with a table known to the operating system, and on RM's passing the function code in an obscure manner. The service eventually fell into disuse, largely because of the time required for the operating system to be loaded, to initialize, and then to execute the command.

Dynamically Available Resources

The first Resource Manager could not accommodate any systems outside of those available on the fixed set of processor bank machines. This meant that resources from CAP, such as terminal sessions or file transfer service, or from RSX, such as bootserver sessions or the terminal information service, couldn't be offered through RM in a manner consistent with processor bank resources.

The resource management system should support the concept of resources as objects offered by components of the distributed system to other network clients. They, or offers to provide them upon request, may become available and be retracted dynamically.

Redirection to Existing Resources

It should be possible to direct clients to existing resources. This was not possible in RM I as there was a one-to-one relationship between client requests and machines loaded. This shortcoming was most painfully realized by users waiting typically from 12 to 25 seconds in the early days for a machine to be allocated, to be loaded with an entire operating system, and possibly for that operating system to run a command. (This time was later reduced to 5 to 15 seconds by improving Ancilla, the machine loading service.) Had there been the ability to indirect users to existing resources, the machines of the processor bank could have been preloaded with popular systems. This would also have improved the three level resource experiment described above.

Without redirection, there could be no shared servers. Initially there was such a shortage of machines that an edit server which could support three sessions was written to ease the situation. Unfortunately, RM could load the server when the first request arrived but could not point any further clients at it.

2.4. Onward

Armed with better knowledge of the requirements of a resource management system, a second Resource Manager was designed and implemented. Experience with the first RM proved invaluable in defining the required functionality, and in formulating a number of test cases against which to judge design and implementation alternatives.

RM I was approximately 3700 lines of Algol68C source code which was compiled for a Z80. It was supported by a combination of Algol and assembly runtime libraries including a simple coroutine system and network protocol drivers. As of this writing, RM II is approximately 10,000 lines of an extended version of CLU which provides processes, monitors, and CLU RPCs. It is compiled for a 68000, and runs on the Mayflower operating system. Most of the functions are available as CLU RPCs, and an increasing number are available as standard Cambridge SSP (Single Shot Protocol) messages. There is a user terminal interface with a simple command executive which provides most of the RPC functions as well as a number of system manager facilities.

3. Resource Description

If resources of a wide variety are to be viewed in a consistent manner, then some scheme for description of these resources must be agreed.

3.1. Definition

In the Cambridge distributed system, control of resources internal to the components of the system is still relegated to the individual components. For example, the management of processor cycles, local memory, and peripherals is the responsibility of the operating systems running on the individual machines. The resources of new interest are the high level ones available among components of the distributed system, such as a filing system session from a file server, a compiler service, a raw machine, or an operating system instance.

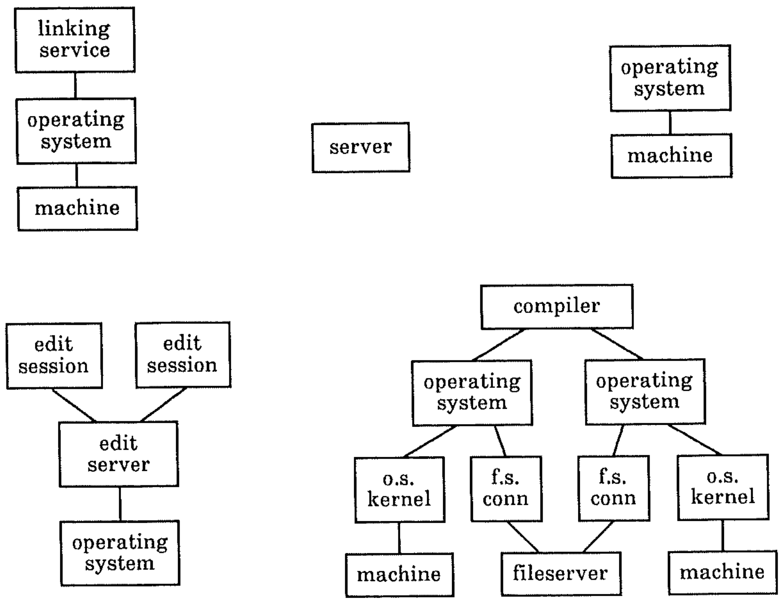


Figure 3.1. Example Resources

A resource may be built upon one or more subresources, with such layering extending for multiple levels. For example, a linking service may be supported by an operating system which in turn runs on a machine [figure 3.1]. Each of these is a resource in its own right, but there are dependence relations among them. These relations form an acyclic graph, which is linear in this example.

Another example resource would be a server, such as a printer server, or an instance of a particular operating system running on a machine. Resources might share supporting components, such as two edit sessions, both supported by an edit server which is running on an operating system. A more complicated example might be a compiler supported by two operating system instances, one for the front end and the other for the back end. Each of these might be composed from an operating system kernel (running on a machine), and a file system connection from a fileserver.

Complexity

One might question whether or not the suggested complexity of resources is necessary. Alternative approaches [Locus83, Accent81, RSExec73] have considered resources as files or processes on remote hosts. However, these have assumed that all communicating hosts will be running the same systems, and that such systems will be bootstrapped external to the mechanisms managing the said resources. This is not the case at Cambridge where communication among heterogeneous machines, languages, and operating systems is held to be important, and the booting of such systems is handled by the same mechanisms that construct client level resources.

Users and other clients are often interested in high level objects, such as compilers and operating systems, rather than low level ones directly available on the network, such as machines and loading services. Unless support is provided for such objects by the resource management system, clients must have the knowledge and provide the mechanism to construct the higher level resources they need from the basic ones.

Resource Information

The information used to describe resources is summarized in figure 3.2, and explained in more detail through the rest of this chapter and in later chapters. Of greatest interest in this chapter is the functionality of resources, specified by attributes.

Resource *factors* are discussed in the "Policy" chapter, section 8.3. They have not been implemented. When satisfying a resource request, these factors are used to decide among the alternatives (existing resources and methods for constructing new ones) which provide the functionality specified by the client.

Allocation information, of which there may be multiple sets if there are multiple allocations of the resource, is discussed in the "Location and Allocation" chapter, section 4.3.

Functionality	*Factors*	*Data for Access*	*Structure*	*Properties for Allocation*	*Allocation Information*
• Class Attribute • Qualifying Attributes	• Quality • Allocation Time • Cost • Comm. Delay	• Network Address • Authentication Information	• List of Supporters • List of Dependants	• To Be Allocated • Reusable • Replenishable • Support Multiple	• Status • Time Left • User (may be multiple allocations)

Figure 3.2. Resource Information

3.2. Attributes

Class and Qualifying Attributes

The functionality of a resource is described by a set of one or more *attributes*. Each resource has a *class* attribute, which gives the basic type of the resource, such as *68000*, *MicroVAX*, or *Mail Server*. Additionally, a resource may have a number of *qualifying* attributes which specify further information, such as a peripheral attached to a machine, a particular processor model, an option provided by a service, a facility available on an operating system, etc. Examples are *floppy disc unit*, *MicroVAX II*, *2 Mbytes* [of memory], or *68000 obj code* [for a compiler]. Several example attribute sets are given below.

Attributes are used by Resource Manager in matching client resource requirements to resources which are available. No interpretation is placed by RM on attributes; rather it compares those possessed by resources with those requested by clients, attempting to provide a resource with a superset of the ones requested. This lack of interpretation supports the generality sought in managing resources. It greatly facilitates adding new attributes, and thus resources, to the system.

Subresource Attributes

When searching for a resource with particular attributes, the attributes of subresources are also scanned. This allows clients to see, and consequently specify, some of the structure of resources they desire. For example [figure 3.3], a CLU compiler running on Mayflower on a 68000 and producing object code for a VAX (i.e. a cross compiler) will have real attributes *CLU Compiler* and *VAX obj code*, but will also appear to have the attributes of its subresources, *Mayflower*, *68000*, and *2 Mb*.

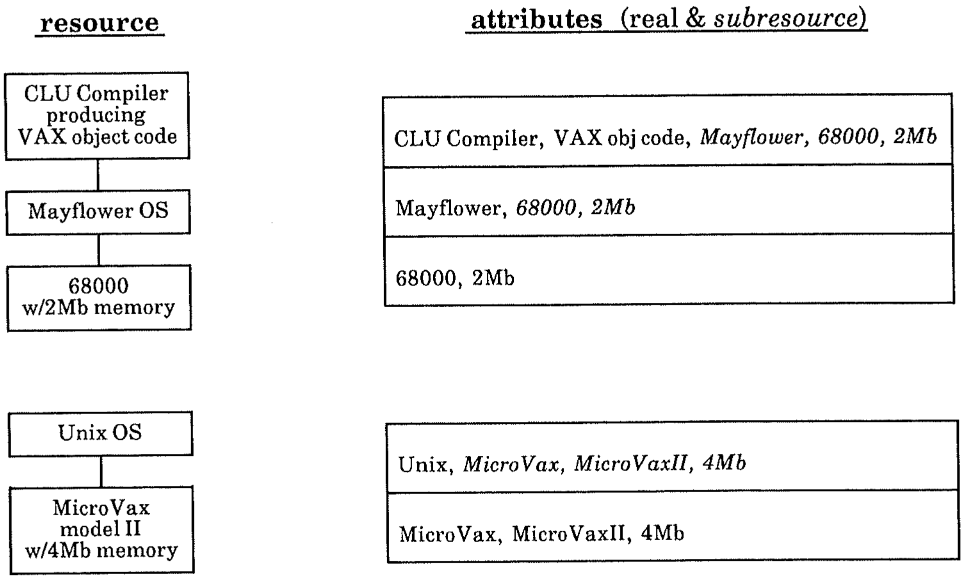

resource
attributes (real & *subresource*)
CLU Compiler producing VAX object code
Mayflower OS
68000 w/2Mb memory
CLU Compiler, VAX obj code, *Mayflower, 68000, 2Mb*
Mayflower, *68000, 2Mb*
68000, 2Mb
Unix OS
MicroVax model II w/4Mb memory
Unix, *MicroVax, MicroVaxII, 4Mb*
MicroVax, MicroVaxII, 4Mb

Figure 3.3. Resource Attribute Examples

Though this approach does not provide unique attribute sets for all different resources, it is unlikely to be a problem in practice because of the nature of resources: it does not make sense for a machine to appear both at the top and bottom layers of a multiple level resource. The advantage is that the description of structured resources is much simpler. A client which requires Unix running on a MicroVAX need only specify the two attributes *Unix* and *MicroVAX* in a resource request, without regard to where they occur in the overall structure. Whether or not this simplification will prove to be a limitation remains to be seen as RM comes into extended use.

The alternative is for clients to describe a desired resource as a tree in which each node is labeled by one or more attributes. This seems needlessly complex as clients are generally interested in the attributes of only a few components. It also requires that clients have knowledge of the full structure of the resource rather than just the components in which they are interested.

Attribute Ordering

There are groups of attributes, such as processor models or memory sizes, which are related by an ordering. Knowledge of such orderings is important to RM so that it can understand that a request for a particular attribute may be satisfied by an attribute which is ***greater*** in the ordering. For example, an *LSI4/30* will suffice for a request for an *LSI4/10*, or *2 Mbytes* [of memory] will suffice for a request for *512 Kbytes*. Attribute orderings change on an infrequent basis and may be entered by system managers. Attributes not belonging to an ordering (the default for new attributes) may not be substituted by or for any other attributes, nor will substitutions be made for attributes specified as "exact" by the client.

An alternative would be to allow a name and value for each attribute. These values could or could not be ordered. Thus the attribute groups above (LSI4 and bytes of memory) would become attributes with associated values (4/10, 4/30 and 512K, 2M, respectively). There might be a limited number of data types allowed for the values: integers, strings, and enumerated types. However, considering that the values for an attribute are generally sparse, that the type for each value would have to be defined and subsequently checked, and that clients would have to specify these name-value structures in each transaction, it is not believed the added complexity would be warranted.

An approach similar to this alternative was taken in Spice [Dannenberg 82]. However, there the set of understood attributes was fixed, the system was single language so there was no difficulty with value types, and the resources of concern were lower level ones (e.g. file pages, seconds of cpu time) for which exact values are more important.

Neither of these approaches to ordering has been implemented.

Representation

In the Cambridge Distributed System, 64 bit ***unique identifiers*** (UIDs) are used for naming. There are permanently assigned UIDs for all objects, including machines, people, servers, and systems [Girling 83]. These UIDs are chosen from a global name space which is centrally adminstered by system managers.

Each attribute is represented internally and communicated around the Ring as one of these 64 bit UIDs. They are read from and displayed to the user as readable strings, using the small server Mercator to perform the necessary mapping.

3.3. Data for Resource Access

As resources are objects available to network clients, they must be accessed via network service interfaces. Often the resource will provide the service interface itself, as in the case of an operating system or mail server. Alternatively, a supporting layer may provide the interface, as in the case of an operating system providing the interface support for a compiler. Resource Manager makes no real distinction: each resource has an associated interface through which it can be accessed.

Address and Authentication Information

To access a resource, a client must know the resource's network address and optionally some authentication information which is expected to accompany requests. The network address will have been uniquely assigned to the resource of concern. At Cambridge, a network address consists of a station, port, and function code. The station will be the machine in which the resource resides. The port identifies the service interface, and is generally constant for that service interface. The function code is used to determine the exact resource. There may be one function code for each type resource, or more than one if several resources of the same type are available.

Authentication information may also be required in a request as a kind of capability. This is generally the uidset for the resource or a 64 bit access token. These are discussed in the chapter on "Protection and Authentication Issues", section 9.4. It should suffice to say that the resource management system will disclose a resource's authentication information only to a client to whom it is allocating the resource (for which privileges will have been checked) or to appropriately privileged system managers.

3.4. Resource Structure

The structure among different resource components is indicated by lists for each resource of all those resources on which it is directly built (its supporters), and all those which are built directly on it (its dependants). These lists are the arcs to and from surrounding nodes in the resource graph. The list of dependants may vary over time, but the list of supporters may not. In the current implementation, a resource is assumed to become unusable if any of its supporters fails. If this is a severe restriction, the resource may acquire its replaceable supporters directly from RM, in which case they will just be resources it uses rather than part of its resource definition.[1]

These lists are used whenever it is necessary to traverse the resource graph. Such uses include: determining which other resources will be affected when a resource fails (section 7.4); scanning subresources, recursively, for attributes (discussed above); altering the allocation status of a resource when a dependant's status is altered (section 4.3); and determining what clients are allowed authentication information for access to debugging interfaces some time after the initial allocation (section 9.4).

[1]The advantage in being part of the resource definition is that, if Resource Manager is constructing the resource, it can ensure that all of the supporting resources are available before instantiating the resource and returning it to the client. Otherwise, the resource might be allocated to the client, only to find that some of the resources it requires are not available.

4. Location and Allocation

4.1. Location

Generally, before a client can access a network resource or service it must determine the location of the resource. This may be done either by broadcasting a request for the desired resource or by requesting its address from some indirection (or resource directory) service.

Broadcasting

A client may broadcast the description of a desired resource and await a response from someone who is willing to provide the resource, or possibly from the resource itself volunteering its use. The client will decide which offer to consider further and there may be additional negotiation between client and provider before the client is allowed to use the resource. This approach has been adopted in Spice [Dannenberg 82]. Such an approach anticipates that providers of resources will always be present to respond to resource requests. It is an effective alternative for distributed systems in which the same operating system runs on all machines, e.g. Locus [Popek 81], or in which servers are static.

Broadcasting resource requests is inappropriate at Cambridge where resources are heterogeneous, multiple level objects. A machine may be loaded with several different operating systems at different times. A client may request a high level resource without having to specify the machine or operating system on which it runs. Thus, a broadcast approach is precluded because an intermediate level resource may not be present to respond to a request for a resource which it supports.

A second difficulty in broadcasting is propagating requests across gateways. Generally, the network protocols will handle the difficulties of broadcasting over bridges (e.g. avoiding infinite propagation around loops), but cannot on their own know whether to forward over gateways. This problem is addressed in Boggs' dissertation on broadcasting in an internetwork [Boggs 82] in which the ability of gateways to broadcast on any directly connected network is exploited to program an expanding ring broadcast.

Indirection

The alternative to broadcasting to locate a resource is to request its address from an "indirection" service. Knowledge of the location of resources in a distributed system is initially distributed. If an indirection service is to be used, this information must be logically centralized (though the service itself may be distributed and replicated).

Binding

Users generally have minimal interest in raw machines, but are more interested in the systems and services which they can run. The establishment of these services and the location of them by users raises several issues of binding. The time at which bindings are made affects the flexibility of the system as well as its complexity. Statically bound services reside in dedicated machines, and their locations are fixed. Dynamically bound services may be brought into existence only when they are needed, and may subsequently disappear.

This spectrum of binding times encompasses a wide range of computing philosophies which have a substantial impact on the habits of use of the system. Toward one end of the spectrum, static network services are bound to machines practically at the start of day. The service to address mappings may be effectively cached by clients. Somewhere in the middle lies the original Cambridge Model Distributed System, in which users are allocated a personal computer for the duration of a login session. All of the user's commands will be executed in that machine. Toward the opposite end of this spectrum is an approach being investigated by Project Mayflower at Cambridge. Commands the user types to the system may be farmed out dynamically to separate processors. The Resource Manager provides support for this full range of binding times.

Static Resources

For resources or services which are static, location information is stable. It may be entered into a table and altered on an infrequent basis by system managers. The service which satisfies requests for "name to address" mappings by querying such a table is generally called a ***nameserver***.

The Cambridge Name Server maps

service name → ***machine address*** + ***subaddress (port and function code)***,

as well as

machine name → ***machine address***.

In programs, services are asked for by service name. Thus, clients ask almost exclusively for the service to address mapping. The Name Server's address is well known and may be written into program code. It is a relatively simple, small server which must be very robust as the Ring world cannot survive without it.

Dynamic Resources

Indirection to dynamically instantiated services requires a varying collection of name to address mappings. There must be some policy to determine who may insert and remove mappings, and some mechanisms for enforcing this. If there are resources intended only for particular clients or groups, it must be possible to restrict the disclosure of their addresses.

In the first version of the Resource Manager there was a sort of "reverse indirection" as the dynamically instantiated resource requested during initialization the address of the client with which it was to make contact. This contact was generally a stream connection into the terminal concentrator controlling the user's terminal. While this was satisfactory for the initial design aims, it could not support sharing or preloading of resources because the resource expected to contact a client only upon initialization.

The second version of the Resource Manager better addresses these requirements for dynamic resources. Its approach to resource location is discussed later in this chapter.

4.2. Allocation

Locating a network resource involves determining an address at which it can be accessed. If the service at that address is willing to handle any request it is sent, then providing location information is sufficient. Clients may approach a static or dynamic indirection service with some description of the required resource, and will receive back an address for the resource. They may then send requests to that address. Any restriction of access is provided by the service or resource itself.

Alternatively, it may be appropriate for the indirection service to provide some restriction of access. This may simply be a binary decision of whether to give the address to the requesting client, or it may also involve tracking the use of the resource and reclaiming it later. In such a case, the resource is *allocated* to the client for some period of time. During this time, the client is assumed to have exclusive access to the resource. When the client returns the resource it may be appropriate to reallocate it, depending upon whether it can recover from the previous allocation.

Because dynamic resources generally are intended to exist only temporarily, there should be some attempt to determine when they are no longer required or have expired, so that they or their supporting subresources may be reclaimed. Additionally, because dynamic resources often are instantiated for a particular client, there should be some concept of allocation of that resource to the client. Resource Manager accommodates resources for which full allocation is appropriate, as well as those for which simple indirection is sufficient.

4.3. The Resource Repository

Resource Manager maintains two data bases about resources available on the network. One, the Resource Repository, contains information about existing resources. The other, the Action Catalogue, contains information about constructing new resources. Following presentations of these two data bases (in this section and section 5.3), there is a discussion of their use in matching client requirements for resources to resources available on the network (in chapter 6).

For each resource in the public pool RM maintains an entry in its *Resource Repository* containing information necessary to manage the resource. RM searches this table when trying to satisfy an incoming resource request.

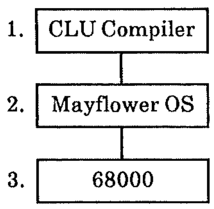
Information About Resources

A Resource Repository entry contains the description of a resource as presented in chapter 3, ''Resource Description''. This information consists of the functionality (class and qualifying attributes), data for access (network address and authentication information), structure (lists of supporters and dependants), properties for allocation (to be allocated, reusable, replenishable, and support multiple), and current allocation (status, allocation time left, and user information) of the resource. Figure 4.1 gives three (related) entry examples for a CLU compiler, Mayflower operating system, and 68000.

Resource Status

The Resource Repository entry includes the current use status, which is used to determine whether or not the resource's current status makes it eligible for a particular allocation. There are three basic states: ***free***, *worm*, and *allocated*. *Free* resources have a status which makes them eligible for any allocation. Resources with status *worm* are slightly more protected. A worm segment may be reclaimed at any time, however Resource Manager will attempt to allocate a resource which is free before allocating a worm one. Additionally, a resource which is running a worm will not be reallocated to run a different worm.

There are two modifiers to these three states. One indicates that a particular resource *supports* a resource of one of the other states. For example, if we have a Mayflower OS which is free, then the machine on which it is running is marked *supporting free*. The Mayflower OS is said to be *preloaded* onto the machine. If this operating system is allocated to a user, the machine is then marked as *supporting allocated*.



	CLU Compiler	Mayflower O.S.	68000
class & attrs:	1. CLU compiler *Mayflower* *68000*	2. Mayflower *68000*	3. 68000
net address:	Carver:20:0	Carver:30:5	Carver:17:3
auth info:	{FF01A783..., 810488A2..., FF053DE8...}	{FF011908..., 19AE0421..., FF0182C3...}	83EA7381...
supporters:	#2	#3	
dependants:		#1	#2
to be alloc:	yes	yes	yes
reusable:	no	yes	no
replenishable:	yes	no	yes
support multi:	no (n.a.)	yes	no
status:	allocated	supp allocated	supp allocated
alloc tm left:	10 minutes		
user info:	G S Murchiston		

Figure 4.1. Resource Respository Entry Examples

The second modifier is that of ***reserved.*** This is an intermediate state which Resource Manager uses to lock resources while it is deciding which actions it must obey to construct a requested resource. Reserved resources are, in general, ineligible for allocation. An exception to this is those resources which may support multiple resources simultaneously.

Allocation Time and User

For each resource which is allocated to a user (or other client) there is an ***allocation time*** remaining. This field is decremented by Resource Manager and upon its expiry, the resource is reclaimed. There is also a certain amount of ***user information*** kept, primarily for listing and (potentially) for accounting purposes.

Recognizing Dynamically Instantiated Resources

Any time Resource Manager has a resource constructed, it will make an entry into its Repository before allocating the resource. Some of the fields for this entry - the attributes, network address, and authentication information - will have been returned by the service which did the construction. The remaining fields either will have been indicated in the information on how to construct the resource, or for fields such as allocation time, will have been provided by the client in the request.

```
Enter_Resource = proc (class:           UID,         % descr of resource
                       attrs:           ARRAY[UID],
                       net_addr:        ADDR,
                       auth:            AUTH_INFO,
                       to_be_alloc:     BOOL,        % properties of res
                       support_multi:   BOOL,
                       reusable:        BOOL,
                       replenishable:   HANDLE,      % resource (in Repos) from whom
                                                     % replacement may be obtained;
                                                     % if null, res not replenishable
                       client_auth:     UIDSET)

                 signals (invalid_uidset,           % of client
                          not_authorized(UID),      % insufficient privilege to enter
                                                    % [resource with] given attr
                          bad_replen)               % bad handle for service from
                                                    % whom to replenish resource
```

Specification 4.1. Procedure for Entering Resource into RM

It is also possible for any service on the network to provide a resource through the resource management system by passing information about the resource to Resource Manager. The procedure for doing this is given in specification 4.1. For example, a file system server could upon initialization register the five file system sessions it was willing to provide with Resource Manager, and replenish these sessions as users abandoned them. This is in contrast to the scheme in section 5.3, where accommodation of a service willing to provide a resource on demand is discussed.

Private Repositories

Resource Manager will maintain private repositories for clients. They may specify certain of these resources to be used in satisfying a particular resource request. In this way clients may utilize the resource construction and management provided by RM to provide a consistent view for both general and private resources. There is a separate mechanism, using privileges, for providing "group" access to resources (section 9.3).

For example, a Mayflower command executive in a user's workstation may farm out any "difficult" commands (e.g. compilation, text formatting) to other processors running the Mayflower kernel. When it determines that it needs a CLU compiler, it may approach RM for one. RM might load a Mayflower kernel and ask it for a CLU compiler, which it would then return to the Mayflower executive in the workstation. Preloading (section 10.1) would keep this from being prohibitively expensive.

It might be that, rather than obtaining a Mayflower kernel from RM for each command, the executive would obtain two or three Mayflower kernels and register them with RM as private resources. Then requests for compilers, etc. could be made to RM specifying that these private resources be used. The executive would still make requests directly to RM, and have the benefit of its construction knowledge, but would be guaranteed use of these two or three machines in its private cache. If the Mayflower kernel was able to support more than one command, then requests to RM beyond the number of machines in the cache would result in multiple commands being executed on each kernel.

Private repositories have not been implemented.

5. Construction

Constructing resources may involve loading code into raw machines, linking code into operating systems, or notifying programs that they are to cooperate as parts of a distributed application.

5.1. Division of Work

The thesis of this dissertation, that resources and the requirements for their management can be viewed in a sufficiently general way as to allow them to be collectively managed, favors providing the functionality for resource construction in the resource management system rather than in external providers of resources. This simplifies the introduction of new types of resources into the system by reducing the functions which must be implemented in a provider, possibly requiring only an addition to a table in the resource management system. It also provides a central collection of information which could be used to guide decisions about construction alternatives.

However, the tasks of resource construction are varied and require specialized knowledge, e.g. of loading protocols or the way in which internal resources of an operating system are combined to produce a user terminal session. Incorporating all such knowledge into the resource management system directly is infeasible because its code would have to be changed each time a new type of resource was added to the system. It is also undesirable because providers of resources may wish to retain some degree of control over those they offer to network clients.

As a compromise, a subset of construction tasks and information has been incorporated into Resource Manager with the remainder left to providers (servers and fabricators) of the resources. The subset was chosen to meet the most common and general requirements.

Management by Providers

Providers may perform some of their own management. For example, a time sharing system such as Unix or the CAP operating system, which was willing to offer sessions through the resource management system, might keep track of available internal memory, the number of sessions currently supported, the number of processes running, etc. They might use these facts to determine whether to allocate another session when one is required. Actually making this decision is a specialized process requiring knowledge of available internal resources and therefore not appropriate to implementation in the Resource Manager.

5.2. Providers of Resources

Servers and Fabricators

Servers are a familiar concept in distributed systems. They offer remote functions such as filing, name-to-address translation, printing, electronic mail, and authentication. Servers are a potential source of resources for allocation via the Resource Manager.

Fabricators augment or combine existing resources to construct target resources. They are a generalization of the Ancillae used by the first Resource Manager, which were used to hide the loading characteristics particular to different processor types and to structure the system. Thus, RM did not have to be changed if a new type of processor was added to the Processor Bank. Instead, a new Ancilla was implemented. This allowed a clean separation of function between machine allocation and machine loading.

A fabricator may be as simple as an Ancilla, loading a memory image into a given machine to produce the target system. Alternatively, it may be sophisticated enough to link the base operating system determined by the target class, and the modules determined by the target attributes, with any libraries resulting from a library scan, and to load this into the given machine.

There are no rigid differences between fabricators and servers. In fact, in the code they are generally referred to collectively as *fab_servs*. However, it is useful to have some intuitive understanding of the notional differences. Both are viewed as capable of providing a target resource when presented with zero or more existing resources. However, fabricators generally require that the client provide at least one resource which is used in the construction. For example, the Ancilla expects a machine which it can load to produce the target system. Servers tend to use only their internal resources to provide the target resource, and they directly support the target resource. Fabricators tend to be associated with the resources they use in construction, rather than the resulting services. For example, a 68000 Ancilla is capable of providing any one of a multiplicity of resources when given a 68000. On the other hand, servers tend to be associated with the resources which they provide, such as an edit server which provides edit sessions.

Interfaces

A request to a fabricator or server [specification 5.1] includes the class and qualifying attributes of the target resource. It also gives its intended use (free/preloaded, worm, allocated). The request also includes the descriptions, addresses, and authentication information of any required existing resources. The server or fabricator is expected to combine these existing resources, possibly along with some of its internal resources, to provide the target resource. Authentication information is included in the request, from which the fabricator or server can determine whether the request is authorized.

```
Res_Request = proc (class:          UID,            % description of target
                    attrs:          ARRAY[UID],     % resource
                    intended_use:   ALLOC_STATE,
                    reqd_resources: ARRAY[RECORD[class:     UID,
                                                 attrs:     ARRAY[UID],
                                                 net_addr:  ADDR,
                                                 auth:      AUTH_INFO]],
                    request_auth:   AUTH_INFO)      % fab_serv or requestor

               returns (ARRAY[UID],   % full set of attributes
                        ADDR,         % network address
                        AUTH_INFO,    % resource uidset (or token)
                        UID)          % dead man's handle

               signals (failed(INT, STRING))  % index of resource which
                                              % failed, and explanation
```

Specification 5.1. Resource Request to Server or Fabricator

The reply includes the attributes, address, and authentication information of the target resource. The attributes may be a superset of those requested. Also included is the dead man's handle, which is part of the mechanism used in monitoring (discussed in section 7.2). Resource Manager uses this interface when obtaining resources from servers or fabricators, but it may be used by any authorized client.

5.3. Actions and the Action Catalogue

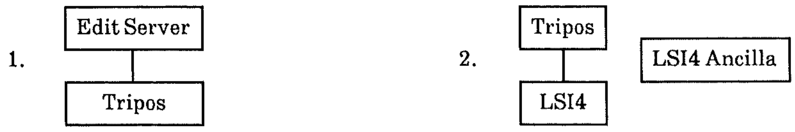
A *resource action* describes how a resource can be assembled or constructed from components with the assistance of a server or fabricator.[1] The action describes how server and fabricator interfaces, as discussed above, can be used, as well as giving static information about the resulting resource (e.g. whether it is reusable). The collection of all resource actions is called the *Action Catalogue*. When RM is trying to satisfy a resource request, it will first search its Resource Repository to find if the resource already exists. If this search fails, it will consult the Catalogue for plans indicating alternative ways of obtaining the resource.

Structure of Resources

Resources have a multiple level structure. It is natural to describe this structure recursively: a resource depends upon a particular set of supporting resources, which may in turn depend upon other resources. Each resource action expresses the structural dependency between two levels of the resource.

Resources supported by several layers are described by more than one resource action. For example, an edit server supported by Tripos running on an LSI4 is described by two actions (see figure 5.1): one indicating the dependence of the edit server on the Tripos, and the other of the Tripos on the LSI4. Additionally, there may be multiple actions required at a particular (sub)level, for example one each to provide the ABC kernel and the filing system session for an ABC operating system.

[1]In the next chapter it is shown how these actions may be combined to form a ***blueprint*** or plan for construction which may then be executed to obtain the target resource.



	result resource	*server or fabricator*	*other reqd resources*
1.	Edit Server	Tripos	[none]
2.	Tripos	LSI4 Ancilla	LSI4
3.	ABC O.S.	ABC Kernel	F. S. Session
4.	Tripos	68000 Ancilla	68000

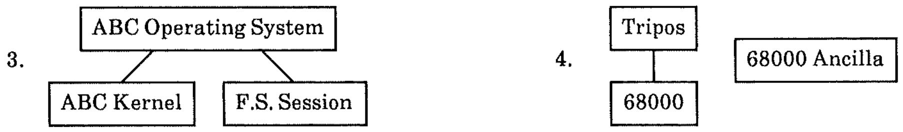


Figure 5.1. Simplified Resource Action Examples

If a resource may be constructed in different ways, then there will be multiple resource actions for it. For example, there are two alternative actions for Tripos: one for LSI4s and one for 68000s.

Method of Construction

Besides indicating the subresources on which a resource depends, resource actions specify a fabricator or server which can construct that resource. If the ***attributes*** for a fabricator or server are specified, then any resource with those attributes will do. In this case, it is Resource Manager's responsibility to find an appropriate server or fabricator. It may do this by looking in the Resource Repository or by searching for an action which it can obey to construct the server or fabricator.

Alternatively, the action may specify a ***particular*** fabricator or server to be used. This would be the result of a service approaching the Resource Manager and offering to provide resources upon request. In this case, the server/fabricator field contains the handle for the particular resource in the Repository.

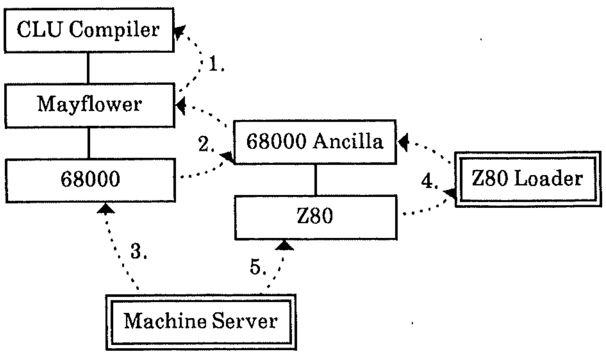
When Resource Manager wishes to obey a resource action, it must first obtain the server or fabricator, and also obtain any required resources listed. Obtaining these resources may be a recursive process. Resource Manager then contacts the fabricator or server, handing it the required resources and requirements for the target resource, and expecting in return the target resource. The required attributes for the target resource may be a subset of those listed as available in the action. This allows the action to indicate the full range of attributes available, while allowing the fabricator or server to provide only those necessary for a particular request. The attributes of the resulting resource may be a superset of those requested. This allows Resource Manager to ask for the minimum attributes it requires, yet to record the actual attributes acquired.

Description of Resulting Resource

For each resource resulting from executing a resource action, Resource Manager makes an entry in the Resource Repository. The attributes, address, and authentication information will be those indicated by the fabricator or server. The attributes will be checked when allocation is made to ensure the client has the necessary privileges. The use status will depend upon the reason for which RM had the resource created.

The remainder of the information for the Repository entry is static for the particular class of resource and is indicated in the resource action. It includes whether the fabricator/server supports the resulting resource (other required resources are assumed to). This is used in constructing the list of supporting resources. It also includes the properties for allocation of the resulting resource (to be allocated, reusable, replenishable, and support multiple) as discussed in section 3.5. These are used in deciding whether the resource is eligible for subsequent allocation or indirection requests.

Figure 5.2 shows examples which include the fields mentioned in the last paragraph. The five actions shown might be used in constructing a CLU compiler "from scratch", i.e. using only primitive resources entered by system managers or the Boot Server (discussed in section 10.2). Assume that only the Machine Server and Z80 Loader exist initially. Action 5 might be executed to



	result resource	*server or fabricator*		*other reqd. resources*	*properties for allocation*			
		attrs or handle	*is supp.*		*to be alloc.*	*reus -able*	*replen -ish.*	*supp. multi.*
1.	CLU Compiler	Mayflower	yes		yes	no	yes	no
2.	Mayflower	68000 Ancilla	no	68000	yes	yes	no	yes
3.	68000	Machine Server	no		yes	no	yes	no
4.	68000 Ancilla	Z80 Loader	no	Z80	no	--	--	--
5.	Z80	Machine Server	no		yes	no	yes	no

Figure 5.2. Resource Action Examples

obtain a Z80 from the Machine Server so that action 4 could be executed to construct a 68000 Ancilla with the assistance of the Z80 Loader. RM would then return to the Machine Server in executing action 3 to obtain a 68000. Following that, it would execute action 2, approaching the 68000 Ancilla and asking it to load the obtained 68000 with the Mayflower operating system. Finally, it would execute action 1, asking the Mayflower operating system for the address of a CLU compiler (interface). The mechanisms for determining and executing this sequence are discussed in the next chapter. (Generally, some of these resources would already exist, either because they are reusable or from preloading, so RM would not have to execute all of these steps at once.)

Creating New Actions

There are resource actions in the Catalogue describing how to build the well known resources in the system. These are generally entered by system managers and are relatively static.[2] The fabricator or server for such actions is usually indicated by attributes, rather than being a specific service, so that any resource with those attributes will do. The procedure for entering resource actions is given in specification 5.2.

```
Enter_Action = proc (class:             UID,          % descr of resource
                     attrs:             ARRAY[UID],   % action produces
                     fab_serv:          ONEOF[class:   UID,
                                             specific: HANDLE],
                     ress_to_acquire:   ARRAY[RECORD[class:  UID,
                                                     attrs:  ARRAY[UID]]],
                     fabserv_supports:  BOOL,
                     to_be_alloc:       BOOL,         % properties of
                     support_multi:     BOOL,         % resource which
                     reusable:          BOOL,         % action produces
                     replenishable:     BOOL,         %     "      "
                     client_auth:       UIDSET)

          signals (invalid_uidset,           % of client
                   not_authorized(UID))      % insufficient priv to enter
                                             % [action for] given attr
```

Specification 5.2. Procedure for Entering Action into RM

It is also possible for any service on the network which is willing to provide a resource on demand to register such an offer with RM in the form of a resource action. For example, the CAP operating system could enter an action to indicate its willingness to consider the provision of terminal sessions or file transfer service upon request. In such a case, the fabricator or server for the action would be the service making the offer. This scheme is in contrast to the one discussed in section 4.3, where a service wishing to provide a resource to the network without having to consider specific requests for it may simply give the resource to RM to manage.

[2]The Action Catalogue is included in the state saved to secondary store, so it will survive Resource Manager failures.

6. Matching Requests to Resources

This chapter discusses in some detail the algorithms used to match client requirements for resources to resources available on the network. It attempts to tie together the implementation covered in the last two chapters, and is rather specific to the functions provided by the Resource Repository and the Action Catalogue.

6.1. Overview

Requests are matched to available resources based on (in order of importance):

- *functionality*,
- *status eligibility*, and
- *factors* differentiating instances with similar functionality.

Available resources include existing ones (those in the Resource Repository) and ones which can be constructed or obtained from existing ones by executing one or more resource actions (in the Action Catalogue).

First, the chosen resource must provide the functionality required by the client. As attributes are used to describe functionality, this means that the resource's attribute set (including subresource attributes) must be a superset of those requested by the client. The class attribute is the most important as it distinguishes the existing resources and resource actions which merit further consideration.

For existing resources, the complete attribute sets are known and can be checked for the required functionality at this stage. For resource actions, the available attributes for the top level resource are known, but attributes for subresources will not be known until later. Thus, this part of the restriction must be deferred. This problem arises because clients see a flat attribute set for a structured resource - though an important simplification for clients, it complicates the implementation considerably.

Next, candidate resources are checked for status eligibility - their intended use (e.g. allocated to client, running worm segment) must not clash with their current use. If a top level resource is being sought, then the intended use will be that which the client specified. If a subresource for a candidate action is being sought, the intended status will be either for allocation (to RM) of a non-supporting fabricator/server or to "support" the intended status for the action resource. Eligibility depends on the current status of the resource and its properties for [re]allocation, as compared with the intended status.

All alternatives remaining after the above restrictions have been applied satisfy the expressed client requirements. Resource management policy must decide among them. The current implementation emphasizes minimum time/delay for allocation. The chapter on "Policy", section 8.3, proposes several factors differentiating resource instances to be compared in making this final choice.

Request and Reply Contents

Basically, requests arrive for a resource specifying a set of attributes. The reply to this request is an address and authentication information for a resource possessing these attributes.

```
Obtain_Resource = proc (class:        UID,
                        attrs:        ARRAY[UID],
                        intended_use: ALLOC_STATE,
                        alloc_time:   INT,
                        client_auth:  UIDSET)

                  returns (ARRAY[UID], % full set of attributes
                           ADDR,       % network address
                           AUTH_INFO)  % resource uidset (or token)

                  signals (not_available)
```

Specification 6.1. Procedure for Obtaining Resource from RM

In more detail [specification 6.1], the request for a resource includes the class and qualifying attributes required, the intended use, the time for which the resource will be required, and the uidset of the requesting client. The class and attributes describe the resource. The intended use indicates whether the resource will be allocated to a user or other client, whether it will run a worm segment, or whether this is just a request for preloading. RM compares this with the allocation status of existing resources to determine their eligibility for allocation for the request. The allocation time determines the limit of time for which the client may retain the resource. The allocation time is ignored if the resource is not to be allocated (i.e. if its address is just given out). The client's uidset is checked for validity and to determine if he has the necessary privileges for the requested resource.

RM's reply contains the full attributes, network address, and authentication information of the resulting resource. The full set of attributes is given because the allocated resource may have attributes of which the client may take advantage though they were not included in the resource request. The network address indicates where the client should send requests to the resource. Generally, the authentication information must accompany requests to the resource as a capability for the right to use it.

6.2. Blueprints

As RM searches for a resource it builds a ***blueprint*** or ***plan for construction***. A blueprint may be a resource node, pointing directly to an available resource in the Resource Repository. Alternatively, it may be an action node, pointing to an action for the resource and a list of blueprints for required subresources.

The Need for Blueprints

Required Resources Available

Resource Manager uses blueprints largely as an optimization. It wants to be fairly certain that a particular plan for construction will succeed before actually beginning such construction.

Suppose, for example, that a BCPL compiler was a sought resource and there were two actions for one (see figure 6.1): the first indicating that a BCPL compiler could be obtained from a Tripos operating system instance, the second indicating that one could be obtained from a Unix operating system

result resource	*server or fabricator*	*other reqd resources*
BCPL Compiler	Tripos	[none]
BCPL Compiler	Unix	[none]
Tripos	Tripos Kernel	Tripos FM sess
Tripos Kernel	68000 Ancilla	68000

Figure 6.1. Simplified Actions for BCPL Compiler

instance. If Resource Manager were to begin obeying the first action, it might discover that it needed a Tripos Kernel and a Tripos Filing Machine session. It could obtain the Tripos Kernel by asking an Ancilla to load an available machine. Once it had obtained this, it would try to obtain the Tripos Filing Machine session. If however, there were none available, and no actions for obtaining one, Resource Manager would back off and try the other action for the BCPL compiler which told it to obtain a Unix instance. Suppose a Unix instance was available in the Repository. RM could approach the Unix instance and request a BCPL compiler.

The attempt to satisfy the first action failed because a Filing Machine session was not available. Unfortunately, Resource Manager had already undertaken communication with the Ancilla, and waited for it to load the determined machine. This would have involved several seconds of elapsed time which was unnecessary because the information was available to determine that that action could not succeed. Blueprints are used to avoid following such blind alleys.

Attribute Requirements Satisfied

Blueprints are also used to ensure that the attribute requirements will be satisfied. Because resource attribute sets have been flattened, it is not possible to tell whether a resource and all of its subresources will provide a superset of the requested attributes until all these subresources are known. Using blueprints, the attributes provided by each subresource are removed from the required attribute set. If the subresource is represented as an action node, then the attributes it contributes will be the intersection of the required attributes and those the action can provide. This intersection is recorded in the action node for inclusion in the request to the fabricator or server. If the required attribute set is empty when the full structure of the blueprint is known, then all of the attributes have been matched.

Description

A blueprint is a tree, each node of which represents a resource (example in figure 6.2). The root node is for the resource requested by the client. The leaf nodes are all ***resource nodes*** pointing to existing resources in the Repository. The intermediate nodes are ***action nodes*** corresponding to actions to be obeyed. The children of action nodes correspond to the required subresources in the action.

Blueprints are recursively assembled, and assembly continues until resource nodes have been reached. The structure of the blueprint reflects part of the structure which the resulting resource will have. The remaining structure is evident in the resources corresponding to resource nodes. Blueprints must contain information necessary for their execution in case they are determined to be satisfactory. Resource nodes simply point to corresponding resources in the Repository. Reservations for these resources will be held by the particular blueprint.

Action nodes contain a pointer to the corresponding action, a list of blueprints for obtaining the required resources, and a list of which of the possible attributes that this action can provide are needed. The list of required resources will include a fabricator or server, and zero or more subresources. An action node would also contain the intended allocation status of the resulting resource.

The example blueprint in figure 6.2 is for obtaining a distributed XYZ compiler in which the front and back ends run on separate machines. In this example, there is only one free 68000 in the Repository so the other is to be obtained from a Machine Server. Once the second machine has been obtained, the Ancilla will be approached to load each machine separately with the PQR operating system, then one of these will be approached (and given the address

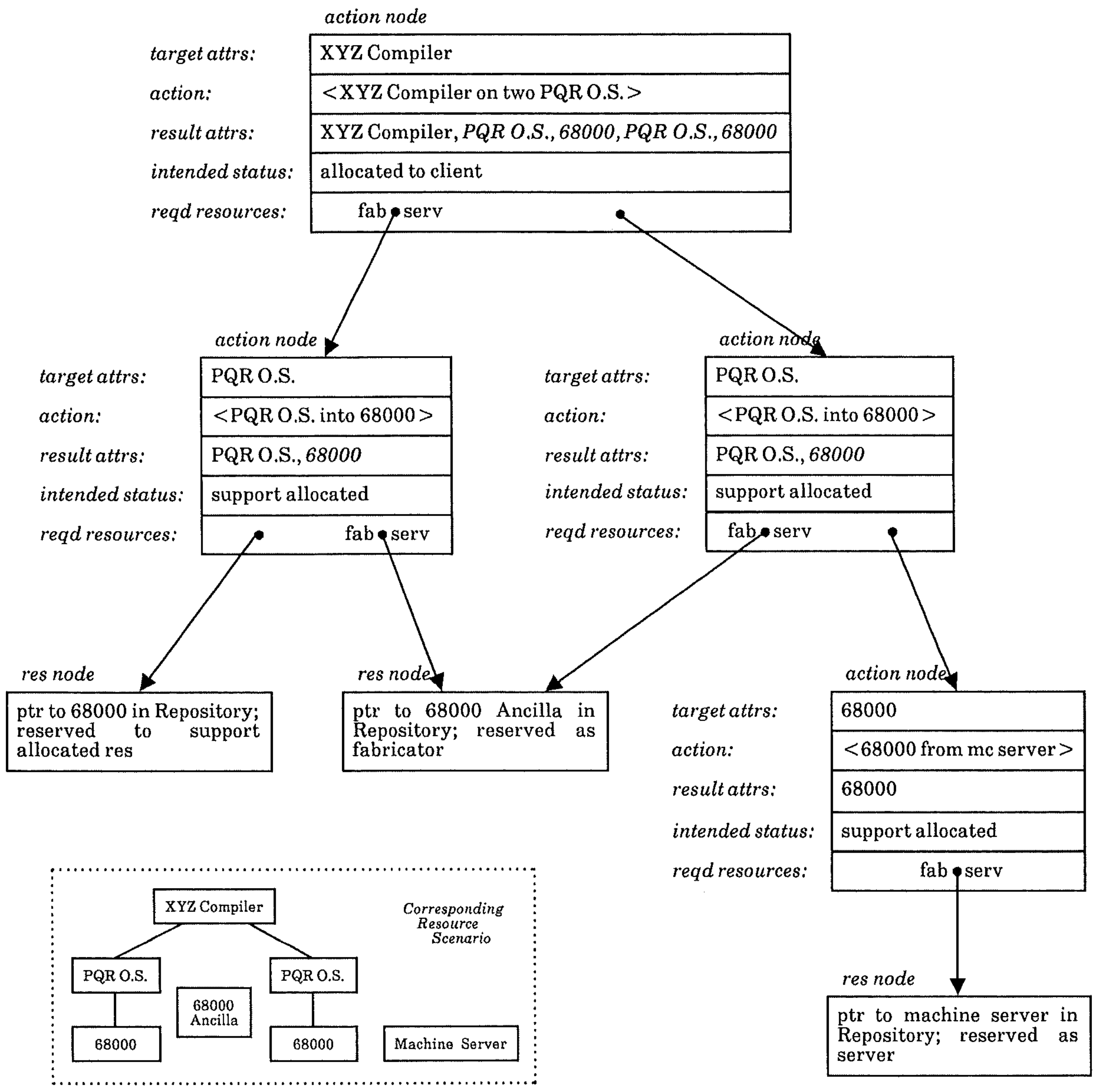


Figure 6.2. Example Blueprint for an XYZ Compiler

and authentication information of the other) to produce an XYZ compiler. There are resource nodes for the existing resources which will be used - the 68000, the Ancilla, and the Machine Server - and action nodes for the remaining components of the structure.

Reservations

As resource nodes are added to the blueprint, reservations are made for the corresponding resource. The reservation indicates the intended status and the request for which it is made.

Reservations are used to prevent a resource from being allocated between the time a resource node is created and the blueprint is executed, as well as from being allocated twice in incompatible ways for the same blueprint. The intended status indicated in the reservation is used to avoid the incompatible uses.

A particular resource may have more than one reservation. For example, an independent fabricator or server may be multiply reserved if it is capable of handling multiple client requests. However, in the current implementation a resource may be reserved for only one intended status at a time.

6.3. Assembling Blueprints

There may be several ways of constructing a target resource given the existing resources and actions. Each of these alternatives has a corresponding blueprint. For example, there may be several instances of a required resource in the public pool and there may be several actions for obtaining it. If each alternative for each node is considered to form a multibranch alternative tree, then iteration of blueprints is accomplished by traversal of this tree. The order of this traversal forms part of the allocation policy and is the subject of the remainder of this section.

Existing resources are allocated in preference to building new ones. Existing resources can always be allocated more quickly than new ones can be obtained for allocation. This provides the quickest response to the user request, possibly at the expense of providing a lower quality resource. If there is a great disparity in the quality of resources in a particular class, and this is important to clients, then an extra attribute indicating the quality may be used so that a client may specify his preference/requirements for quality. It is not believed this will be a serious problem.

The policy of allocating existing resources in preference to building new ones also avoids the unnecessary proliferation of resources of which there are already usable instances, even though it may be possible to construct slightly superior ones. This is important because new resources can often be created only by reclaiming others, which could lead to a form of thrashing if successive requests had alternate intentions for a particular (superior) subresource. Different fabricators which run on the same type of machine are particularly susceptible to this as fabricators are generally idle and thus eligible for reclamation.

The remainder of this section discusses the alternative blueprints tree and the algorithm for its traversal. The reader who does not require more detail may wish to skip to section 6.4.

Alternative Blueprints Tree

The tree whose traversal produces alternative blueprints (which are themselves trees) has both alternative and conjunctive nodes. Wherever a resource is required, either by the client or for use in a resource action, there is an *alternative* node - a node whose branches are alternatives from which one must be chosen. Wherever there is a resource action, and one or more resources must be assembled, there is a *conjunctive* node - a node whose branches must all be satisfied. The leaf nodes of the tree represent existing resources in the Repository.

Figure 6.3 gives an alternative blueprints tree for a CLU Compiler. It is an extension of the example discussed in section 5.3 (pp. 38-9), which the reader may wish to review. This is a fairly restricted example in that there is only one action for each resource class. If there was an action for a CLU Compiler on Unix, then there would be an additional branch from node *A* with its corresponding subtree.

A blueprint is basically a partial image of this tree which, from the root, follows one of each of the branches from alternative nodes and all of the branches from conjunctive nodes until leaf nodes are reached. The image for the example discussion in section 5.3, in which only the Machine Server and Z80 Loader existed initially, is traced by the dotted line. Alternative nodes with more than one branch (*A*, *D*, *H*, *M*, and *S*) give rise to multiple blueprints. As such, they do not appear in the blueprints. Conjunctive nodes (*C*, *G*, *J*, *P*, and *X*) appear as action nodes in the blueprints, and leaf nodes appear as resource nodes.

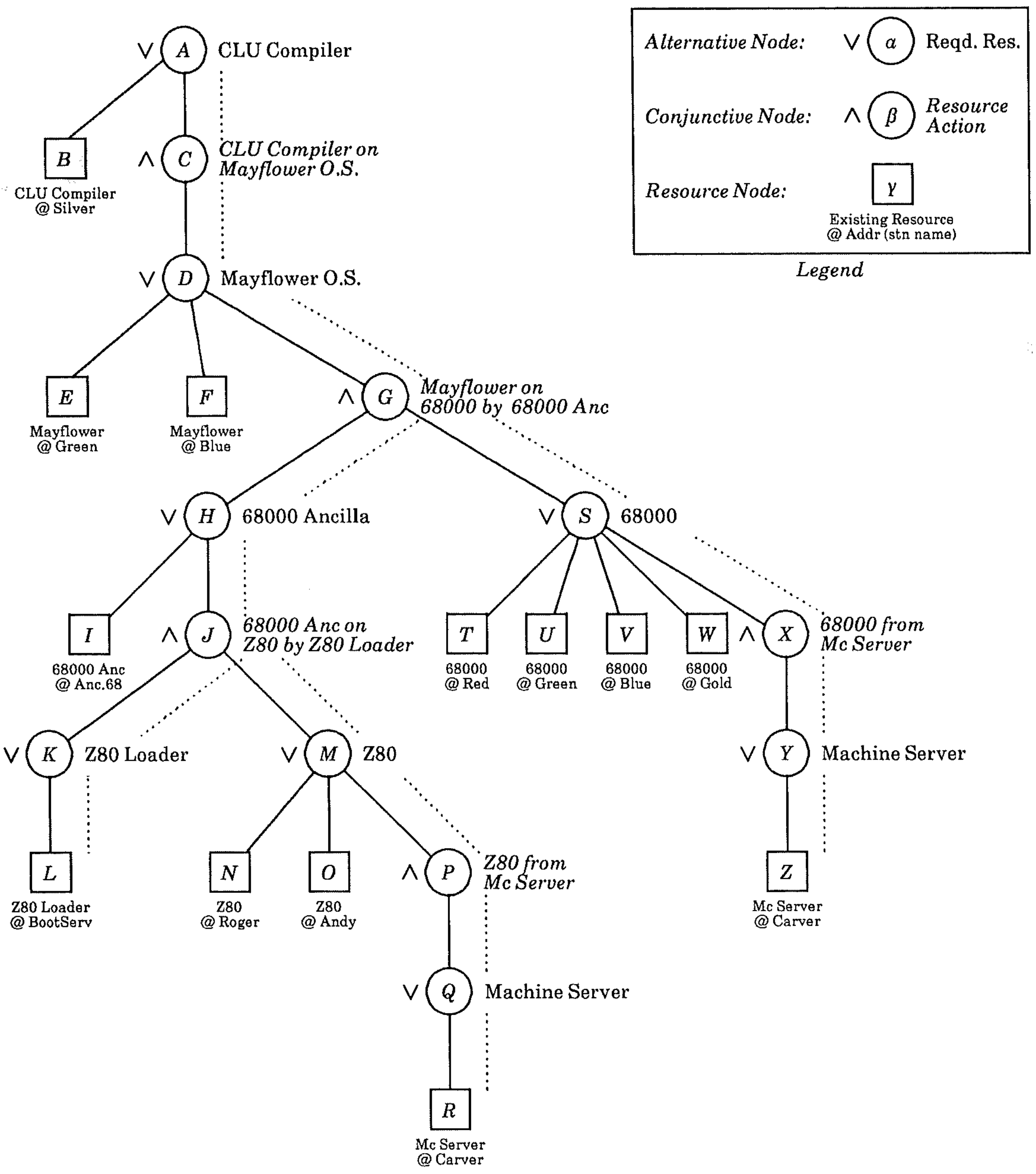


Figure 6.3. Example Alternative Blueprints Tree

Traversal Algorithm

Walking the tree to produce blueprints is not a standard traversal problem (in which the goal is to visit each leaf node exactly once) because of alternative nodes. For each branch of each alternative node appropriate parts of the rest of the tree are scanned, resulting in multiple visitation of many nodes. For example, node *S* will be visited once for each alternative of nodes *H* and *M*. However, the policy to use existing resources before constructing new ones does impose a form of breadth first traversal - we would like to descend as few levels as possible since each walk through a conjunctive node implies construction.

At each level, all combinations of resource branches are considered before any action ones are. Because this must apply *across* a level, it cannot be accomplished by recursive application of single autonomous scanning routines at each node which simply consider their alternative resource branches before their alternative action branches. Instead, control is applied from parent conjunctive nodes to make two separate passes over each alternative node for its resource and action alternatives.

Considering existing resources before resource actions ensures, for example, that an existing Mayflower system (resource node *E* or *F*) will be used in preference to loading one on a 68000 (action node *G* and resource node *T*, *U*, *V*, or *W*), but it says nothing about which of the existing 68000s is to be preferred. The order in which resource and action alternatives are considered is discussed below.

Order of Resource Nodes

Resources in the Repository are kept in ordered lists based upon their eligibility for allocation. There is one list for each class. When scanning for resource nodes, each element in the list for the required class is simply considered in turn, until either the resource is acceptable or the allocation potential of the remaining resources is lower than that which is required.

Eligibility for allocation is determined by the current allocation status of a resource, including reservations, in conjunction with its properties for allocation. The ordering of eligibility attempts to allocate first resources which are free, then reusable/replenishable ones which have been preloaded (that is, are supporting free resources), failing that reusable/replenishable ones which are running worm segments, and finally ones which are already supporting resources allocated to clients and can support multiple resources. In the previous example, it is this ordering which would cause a requirement for a 68000 to consider nodes *T* and *W* (assuming they are free) before *U* and *V* (which are preloaded, assuming the Mayflower systems of nodes E and F are free). Because of reservations, this ordering is only partial: one resource might be eligible for a particular intended status while another is not, yet vice versa for a different intention.

In addition to checking allocation eligibility of the resource, RM must ensure that the client has sufficient privileges to be allocated a resource with the attributes possessed (discussed in section 9.3).

Minimal Search Time

The reason for ordering the resource lists was to keep the search time for a resource of a particular class to a minimum. With ordered resource lists, the first resource with a status eligible for the intended status is the "most eligible" resource available. If that is unacceptable because it lacks the required attributes or has attributes for which the client does not have the necessary privilege, then the next resource with a status eligible for the intended status is the next most eligible, etc. The ordering seemed particularly important in view of the fact that the number of times a scan is repeated is exponential with the number of components of the target resource.

Whether or not the decision to keep resources ordered was correct is unclear. The ***status*** cluster, which incorporates the ordering functions, was greatly complicated as a result. It is the largest and most complex cluster in RM. Changing a resource's status requires changing its position in the list. Considering that in practice the number of resources of a particular class and the number of components in a target resource are often small, the time taken to consider each resource for each search iteration may have been acceptable. However, much of the current complexity is in calculating the factors to compare resources for eligibility. This calculation would still be required, though its frequency would be slightly less.

Order of Action Nodes

RM considers actions in the order in which they appear in the Action Catalogue. This order is static and is preset by system managers to yield the more desirable resource assemblies first. No attempt was made to dynamically reorder actions because it is in general not possible to determine factors such as fabrication time and quality of resulting service. There is some discussion of how this might be done in section 8.3.

An Alternative to Special Traversal

The traversal of the alternative blueprints tree is designed to produce the "best" blueprint first so that it may be executed, and only if it fails need the next best blueprint be produced, etc. An alternative is to create all of the blueprints for a particular resource, and then decide among them. This was considered infeasible because of the complexity it would introduce in reservations and of the large number of blueprints for even a relatively simple resource.

For example, suppose that for a desired resource of which there are no free instances in the Repository, there is one resource action indicating it can be constructed from two 68000s by a 68000 Ancilla. Assume there are eight eligible 68000s, one 68000 Ancilla, one resource action describing how to construct a 68000 Ancilla from a Z80, and two eligible Z80s. There would be 56 combinations of 68000s (8×7) times 3 choices for an Ancilla, or 168 blueprints. The reservation system would have to understand that a particular 68000 could be reserved only once within a blueprint but multiply by different blueprints for the same client request, though not by different blueprints for different client requests.

6.4. Execution

Once an acceptable blueprint has been completely assembled, RM attempts to execute it from the bottom upward. RM confirms the reservations for resource nodes then contacts the fabricator or server passing it information about the existing and required resources, and expecting information about the required resource in return. For this newly created resource, an entry is made in the Repository. If the resource is a top level resource, it is then allocated to the client.

Failure Recovery

If blueprint execution fails because a resource fails during fabrication, then resources which have already been acquired for the action being obeyed are released. Otherwise, the failure was in confirming a reservation for a resource. In either case, any parts of the blueprint depending on the failed resource are dismantled. The particular resource which failed is added to the bad resources list for this request so that no attempt will be made to reuse it at a later stage in the current request.

At this point, there will be a number of suspended resource and action node scans, one for each resource required at each level. These are successively resumed and will check to see if they yielded the offending resource, in which case they will move on to the next alternative. If they did not yield the offending resource, then they will terminate as some earlier node in the tree has yielded it and must be replaced. When the scan yielding the failed resource is reached, then reconstruction of the blueprint beyond that point will be attempted. If this attempt is successful, RM will try to execute the new blueprint in the same manner.

7. Monitoring and Reclamation

7.1. Requirements

Once created, a resource should be monitored so its failure can be detected. Once allocated, several additional changes in status should be noted: the client may inform the resource management system that it has finished with the resource; the overall allocation time for the resource may expire; or a higher priority request for the resource may arrive (leading to preëmption). All of these events indicate that the resource will shortly be reclaimed. Notification of these events and of the actual reclamation should be provided for interested parties.

The user or designer may be interested in the failure of a resource so that the resource can be debugged. Other components of the client task for which the resource was allocated may be interested in its failure, expiry of its overall allocation time, or its preemption so they can reconfigure or can replace the resource. The resource management system is interested in these events so that it can adjust its internal tables and cause the resource to be reclaimed at the appropriate time. The provider of a resource may be interested in its failure so that it can rectify the problem, possibly calling it to the attention of system managers. The provider may also be interested in an indication of reclamation so it can reset or replace the resource.

When a resource is allocated, an allocation time is specified by the client after which the resource may be reclaimed. This is primarily to recover resources which the client has neglected to return or whose failure was not detected when it occurred. Depending on the type of resource, the resource management system may make it available for reallocation or cause it to be reclaimed.

Mechanisms to support monitoring and notification were designed for Resource Manager. Because they were felt to be useful elsewhere in the distributed system, two small servers, known as the Aliveness Server and Event Notification Server, were postulated to support these abstractions. These servers currently reside in the same machine as Resource Manager though their CLU RPC interfaces (and similar SSP interfaces) are available as network services. Communication with them by Resource Manager and between the two servers is via RPCs.

The remainder of this chapter describes the mechanisms for monitoring and notification, as well as how they are used in reclamation and preemption.

7.2. Aliveness Monitoring

The *Aliveness Server* keeps track of which services or resources are running, or stopped but of interest to another service (e.g. a debugger). This is done using a ***dead man's handle*** mechanism in which a small refresh time (generally on the order of minutes) is associated with each object. The object, or a client acting on its behalf, must contact the Aliveness Server within the time limit and request that this time be extended. If the timer expires, the object is assumed to have died and an ***expire*** event is sent to the Event Notification Server indicating expiry of the given object.

Once an object is believed to have expired, the Aliveness Server will wait a nominal time (two minutes) and, if the refresh is not resumed (e.g. by a debugger), will then issue a ***reclaim*** event for that object. The time between the expire and reclaim events may be used for cleanup or recovery by an interested party (the user, a parent task, etc.).

Starting a Dead Man's Handle

```
Start_DMH = proc (handle:       UID,                 % THE dead man's handle
                  name:         SEQUENCE[STRING],    % name of the associated object
                  init_time:    INT,                 % initial refresh time
                  client_auth:  UIDSET)              % as authenticator of "name"

            signals (handle_in_use)
```

Specification 7.1. Aliveness Server Procedure for Starting Dead Man's Handle

A dead man's handle is started [specification 7.1] by presenting a 64 bit UID which will act as the handle to be shaken, a sequence of strings naming the object (e.g. resource class and address), an initial timeout, and a uidset for the client starting the handle. The handle should be random/secret as it will act as a sort of capability for extending the refresh time or forceably expiring it. The sequence of strings naming the object will be sent to the Event Notification Server as arguments for the ***expire*** and ***reclaim*** events. The authenticity of this name is guaranteed by the client's uidset.

7.3. Event Notification

The *Event Notification Server* will notify interested parties when an event occurs (e.g. a resource has failed to refresh a dead man's handle). The concept of asynchronous cross process signals as it occurs in operating systems is not dissimilar to the Notification Server events: issuing a "wait" is similar to registering interest in an event, and the resulting "notify" is similar to the notification callback.

Registering Interest

Any client may register interest in an event by giving [specification 7.2]: the event's name and arguments, optionally the service from which the event must be triggered and the service which must have provided the qualification (if different from the triggering agent), a timeout for the interest, an SSP call or CLU RPC to be made when the event occurs, and an indication of how imperative it is that such notification succeeds (***try*** *once*, ***try reasonably***, or ***try*** *until success*).

```
Register_RPC = proc (on_occurrence: UID,                   % the event of interest
                     args:          SEQUENCE[STRING],      % qualifiers on that event
                     triggered_by:  UID2,                  % who must trigger it
                     args_by:       UID2,                  % who must have given args
                     within:        INT,                   % mins till interest expires
                     notif_factor:  IMPERATIVENESS,        % how imperative notif is
                     rpc_proc:      REMOTEPROCTYPE(TOKEN[ANY]),
                     rpc_arg:       TOKEN[ANY])            % RPC for notification
```

Specification 7.2. Notification Server Procedure for Registering RPC Interest

When an event is triggered, the calls for all matching interests will be made asynchronously. Because anyone can claim that events have occurred, a client registering an interest may indicate the party who must trigger the event. Then only authenticated claims by the specified party will cause the RPC or SSP call to be made. (If the arguments describing the event originated from a third party, that party may also be specified. For example, a client may be interested in the event {*expire* "Tripos" "@Green:30:1"}, indicating that the Tripos on the machine named Green has failed to refresh its dead man's handle, only if it comes from the Aliveness Server, and only if the arguments "Tripos" and "@Green:30:1" came from Resource Manager.)

Naming Events

Events are named by a 64 bit unique identifier (representing e.g. *expire* or *reclaim*) and a sequence of string arguments giving successive qualifications of that event. Strings were chosen as the type of the qualifying arguments because of the variety of data which must be represented - the (mapped) UID for "Tripos", the station/port/function code address "Green:30:1", the month "Apr", and the number "1984", etc.

When an event occurs, its name and arguments are checked against registered interests. There is a match not only if the name and arguments are equal, but also if the names are equal and the interest arguments are a prefix of the event ones. Thus, interest in the reclamation of any 68000 {*reclaim* "68000"} would be triggered by the event {*reclaim* "68000" "@Red:23:0"}, and interest in {*clock_time* "1984" "Apr" "02"} would be triggered by the event {*clock_time* "1984" "Apr" "02" "00" "00"} (*clock_time* events are generated by the Event Notification Server every minute).

In addition to the communication of public events, such as *clock_time*, *expire*, and *reclaim*, the Event Notification Server may be used to communicate private events by using a random/secret UID and not publishing that UID to the world at large.

Notification Calls

The interested client may have specified an SSP call (address, and arguments as a sequence of bytes) or an RPC (remote procedure taking a single

token[1] argument, and the value for the token) to be made if the event occurred. Though this SSP or RPC will generally be made back to the interested client, it may be destined for a third party. For example, the File Server may register an SSP destined for RM (to initiate a File Server garbage collection) when the clock event for 2AM the next morning occurs. If authentication is involved, e.g. the SSP/RPC arguments include a uidset, it is up to the client to ensure that the authentication information remains valid until the call is triggered.

7.4. Use for Resources

When Resource Manager obtains a new resource - receiving it from a provider, requesting it from a server, or having it constructed by a fabricator - it initiates a dead man's handle in the Aliveness Server. (Part of) the resource's authentication information is used as the handle, and the resource's class and address (e.g. "Tripos" "@Brown:30:1") are used as the name. The resource or its server is responsible for refreshing the dead man's handle timeout. If the timeout expires, the Aliveness Server will notify the Event Notification Server of the *expire* event for the resource, and two minutes later of the *reclaim* event. Resource Manager registers interest in the *expire* and *reclaim* events for each resource.

Intent to Reclaim

Because resources are structured, the failure or reclamation of one resource may affect others. If the *expire* event for a resource is triggered, RM will trigger the *to be reclaimed* event for the resource, as well as its dependants and any supporting resources which are not reusable. Any party interested in the fact that a resource will shortly be reclaimed should have registered an interest in this event. This may include the user or designer who will take up the refresh of the dead man's handle while he debugs the resource, or if the resource is a component of a distributed task, a parent, sibling, or child which wishes to commence a cleanup or recovery operation.

[1]The *token* cluster associates CLU objects with one word tokens which may be passed to remote machines, and the objects retrieved when the tokens are returned.

This same sequence of actions will be triggered by RM if the overall allocation time for a resource expires. In certain cases it may be appropriate for the resource itself to register interest in when it is to be reclaimed. For example, an operating system might do this so it could notify the user of imminent termination of his session.

Reclamation

Resource Manager basically reclaims resources by throwing them away. If a *reclaim* event for a resource is triggered, RM will delete the entries for the resource, its dependants, and its non-reusable supporters from the Repository. It will also go to the Aliveness Server and force the *reclaim* event to be triggered for the dependent and supporting resources. (It can do this because it knows their dead man's handle values.) This results in the notification of parties interested in the reclamation of these resources.

Providers of resources may wish to register interest in the *reclaim* event. For example, a machine server may wish to reset the physical machine and note in its tables that this machine is available to RM. A filing system server may wish to close channels and reclaim memory allocated for a session resource and create and enter a new free session into Resource Manager.

7.5. Use in Preemption

Preemption is the forceable reclamation of a resource due to a higher priority request for it arriving. The advantages and disadvantages of preemption are discussed in the chapter on "Policy", section 8.3. Although no policy for preemption has been decided or implemented, its possible implementation is discussed here as it is a form of reclamation.

The Event Notification Server provides the basic mechanism needed for notification of preemption. When Resource Manager decided to preempt a resource, it could generate a *preempt* event indicating which resource it intended to preempt. It would then wait some nominal time (possibly 30 seconds) before generating a *reclaim* event and actually reclaiming the resource. Any service which wished to perform a cleanup operation for a resource should have registered an interest in that resource's preemption. This would allow Resource Manager to notify all interested parties with one action that a resource was about to be preempted.

Examples of interest in preemption and the resulting actions which might be taken follow. An operating system, such as Unix or Tripos, might register an interest in its own preemption. If it received notification that it was about to be preempted, it could notify the user that he should clean up and logoff within 30 seconds. In the case of a compiler service, notification of imminent preemption might result in a consistent intermediate result being prepared and written to disc. If the resource was part of a distributed task, its parent might register an interest so that it could reallocate the resource's subtask, possibly at a later time. In the case of a worm segment, no interest would be registered as a worm is capable of recovering from a failure/reset at any point of one of its segments.

8. Policy

Resource management policy guides decisions concerning the recognition, construction, allocation, and reclamation of resources.

8.1. Introduction

The thesis of this dissertation is that sufficient commonality exists in the requirements for the management of resources that they can be collected effectively into a resource management system, thus minimizing the requirement for resource management mechanisms in individual components. The implementation of resource management policy severely tests this thesis.

In operating systems, policy for the management of internal resources such as memory and processor cycles, is generally highly tailored for efficiency: the resources must be allocated quickly as the time for which they are required is often short, and there must be small space overhead as the resources themselves are relatively small. Such tailoring is less important when the resources are of a higher level, and thus inherently larger and held for longer intervals.

Provide Several Policies

The policy requirements for different classes of resources will vary. Rather than providing a single policy encompassing all of these requirements, several policies are provided, with the application of appropriate ones to specific classes. If these general purpose policies are insufficient for some particular class of resource, then a separate server may be provided which enforces the required policy for that class.

Additionally, individual resources or the servers providing them may enforce a limited form of policy of their own, e.g. checking access control lists, placing a limit on the time or memory the user is allowed, etc. However, these policies may be based only on local information (as opposed to information about the allocation of all of the resources available on the network) unless other resources or servers are contacted explicitly.

In this research, the primary emphasis of implementation was on providing the basic mechanisms for a resource management system rather than on deciding and implementing management policy. Inevitably, to make Resource Manager usable and to determine its viability as a resource management system some policy has been enforced. A presentation of this implementation is followed by a speculative discussion of alternatives and extensions.

8.2. Current Implementation

Allocation and Construction Order

Currently, existing resources are always allocated before an attempt is made to construct new ones. This policy emphasizes fast response time to the user's request as the construction of resources may take considerable time.

The ordering for allocation of existing resources is based upon an ordering of status: *free* < *preloaded* < *worm* < *user*. Whether a resource is eligible for allocation depends on its status as compared with the intended status expressed in the resource request, and the indication of whether or not it is reusable and replenishable. For example, a resource running a worm segment would not be used for preloading though it would, if it was reusable, be used to support a resource to be allocated to a user.

The policy for the sharing of resources is that a resource will be allocated on a shared basis only for an intended status the same as its current status, and only if no more appropriate resource exists. The implementation uses the above ordering and information associated with each resource indicating whether it may support multiple users.

The order of consideration of actions for constructing a resource of a particular class is static. It may be adjusted by system managers, either to provide the higher quality construction first or to delay the allocation of scarce subresources. No attempt is made to dynamically reorder actions.

Allocation Restrictions[1]

The restriction of allocation of resources of a particular class or with particular attributes is accomplished by associating privileges (registered against clients in the AOT Privilege Manager) with the class or attribute to which access is to be restricted. Two privilege lists are kept for this allocation restriction. One associates the right *to have* a resource with a particular class or attribute allocated directly. For example, this list might have *research staff privilege* associated with the class of an operating system to be allocated only to research staff. The other associates the right *to use* a resource with a particular class or attribute as a subresource of the resource to be allocated. For example, the attribute *Mayflower machine* (for machines belonging to the Mayflower group) might be registered in this list as requiring clients to have *Mayflower privilege* before one of these machines would be used in constructing a resource for them.

Restriction to the authentication information allowing access to the debugging interface for a resource is discussed fully in section 9.5, "Capabilities for Debugging". The currently implemented compromise is always to allow a user access to the authentication information to debug a resource which he has been allocated. Additionally, a privilege list is kept associating privileges with attributes so that the maintainer of a particular service or resource may debug it at any time.

8.3. Alternatives and Extensions

The policy of allocating existing resources before constructing new ones is not always desirable. A low quality existing resource may be allocated in preference to constructing a much higher quality resource though the user has to wait for its construction. As suggested previously, if there is great disparity between the quality of different instances of a resource, then it may be appropriate to have *low quality* and *high quality* attributes. However, the suggestion of resource factors given below presents a better continuum.

[1]A detailed discussion of restriction by privileges is given in chapter 9, "Authentication and Protection". An overview is provided here as it is part of the implemented policy.

The policy for sharing is that no resource will be shared if there is another free instance of it. This could result in several instances being allocated to single users though the resource could reasonably support more, thus unnecessarily consuming a number of subresources to the exclusion of other requirements for them. At the other extreme, if there is only one instance of the resource then it will continue to be shared (possibly overloading it) when more instances could be constructed.

Caution and Motivation

When considering alternatives and extensions to the current, relatively simple policy it is important to remember two points. First, no amount of resource management policy can compensate for a lack of resources - scarce is scarce. The problems introduced by having too few resources to operate "reasonably" are not being pursued in this research. Second, ***maximum*** utilization of the system is ***not*** a primary goal due to the effort required to achieve it and to the decrease in responsiveness to users which would result.

Overcoming Personal Computer Limitations

A major contribution of the personal computer approach to distributed computing, including the original Cambridge Model Distributed System, has been that throughput of the system has not taken priority over fast and predictable response to users. This has been accomplished by not sharing machines - a user's tasks may affect each other but generally not those of other users. However, responsiveness is still limited by the single machine - witness the delay in recompiling and linking a medium to large program. The processor bank approach attempts to overcome this limit by providing easy access to multiple machines when tasks demand. Unfortunately, demand would probably always outstrip supply if the personal computer approach was simply extended to multiple machines, allowing users to monopolize a number of machines with long-lived, largely idle sessions.

Transaction vs. Session Programs

Two patterns of program behavior are of interest. ***Transaction oriented*** programs, such as compilers, linkers, and text formatters, are short-lived and c.p.u. intensive. It is reasonable to assign an entire machine to each such task and to allow it to run to completion without intervention. ***Session oriented*** programs, such as text or graphics editors, APL sessions, and Lisp sessions, generally run for long periods of time and are interactive. Long existence and low c.p.u. utilization dictate that a session program be run in the user's work-station or in a shared processor bank machine.

If session programs can be used primarily to collect task state and to wait for user input, and can execute [remote] transaction programs for their intensive computation, then performance on shared machines should be acceptable. However, this may not always be feasible, either because of the amount of state which would have to be transferred to the transaction program, or because the program was imported from another system and is not easily modified to achieve the required behavior.

The current implementation of Resource Manager can cope with transaction versus session resources. For example, a compiler to run on Unix might require a resource with attributes {***Unix, for transaction***}, while an APL or login session would require {*Unix, for session*}. Resources with the *for transaction* attribute would have the *support multiple* property set to FALSE so they would not be shared, while resources with the ***for session*** attribute would have it set to TRUE. Further support must be provided to decide whether to add another session to a shared resource, and if so to which one, or to construct a new resource because the existing ones should not be more heavily loaded.

Resource Factors

Several ***factors*** may be used to indicate the variance in ***instances*** of resources of a particular class. Applied to existing resources and actions for constructing new ones, these factors could be used in determining the most appropriate way of satisfying a resource request. These factors include:

- *quality* - a somewhat nebulous concept (difficult to quantify) used to order instances based on their performance or usefulness; particularly a problem for shared resources for which it varies even after allocation;
- *allocation time* - the approximate time which will elapse between deciding how to obtain the resource and its being ready to allocate;
- *cost* - the charge for use of the resource (for accounting); and
- *communication delay* - an indication of the "remoteness" of a resource accessed through bridges or gateways.

It should be stressed that these factors need to be only approximately correct; it is their relative (rather than absolute) values which are important.

Determination of Factors

The calculation of factors has been kept simple and uniform as it must be made at allocation time. Factors filter upward from basic resources to client level ones. At each level, contributions for that level are added to the values from supporting resources (diminished according to their importance).

For an existing resource, the factors are recorded in its Repository entry, having been calculated when the resource was created. For example, the allocation time for an existing resource is zero; the cost for use of a machine might be a constant known by the machine server.

For a resource action, the contributions to each factor for that layer are recorded in the Catalogue entry. For example, the time for execution of an action (either statically approximated, or dynamically determined by noting the time required for its execution in the past) would be its contribution to the allocation time factor. Also recorded in the Catalogue entry are fixed percentages indicating the contribution to each factor from each subresource. These reflect the "importance" of the subresources for each factor - a supporting resource which played little part in the overall resource would contribute little in terms of quality, while its allocation time would still have to be considered in total. Once the subresources are known (when the blueprint has been constructed) the value of each factor is calculated as the sum of the layer contribution and the inner product of the percentages with the values for subresources.

Use of Factors

When considering alternative blueprints for a resource, some linear combination of the factors, whose formula was constant for all resources, could be used to decide among them. Very simple client restrictions (e.g. "minimum allocation time" or "no more than this cost") might also be allowed. Because it may be infeasible to assemble all of the blueprints and then compare them [section 6.3], it may be necessary to prune the tree by choosing the "best" branch at each alternative node. This would introduce problems with local optimizations and with backtracking if the chosen branch failed during blueprint execution.

Consideration of the allocation time and quality factors would replace the current policy of always allocating existing resources before constructing new ones. Consideration of the quality factor, possibly along with *optimum* and *minimum* values might provide a form of load balancing, replacing the current policy for sharing. The optimum and minimum values might vary dynamically with overall system load to ensure higher quality resources during periods of low utilization. Resource factors merit further consideration.

Policy Language

An alternative to providing a small number of fixed policies which must apply to all resources is to introduce a policy language to express predicates which must remain satisfied when a particular class of resource is to be allocated. The current allocation states of resources, including which resources they support, whether they are shared, etc. would be available so that policies spanning several types of resources could be implemented. This would allow policy such as "at most three 68000s are to run student systems at any one time" to be implemented. Time of day, day of week, current system load, etc. might also be available.

This scheme allows a form of individually tailored policy for resources which can change dynamically without altering Resource Manager code. All the required information is currently available in Resource Manager, but how much should be made available in the policy language and the form which it should take is unclear without further consideration. As a policy language would require major effort, it is hoped it will not be necessary.

Preemption

Preemption is the forceable reclamation of a resource due to a higher priority requirement for it or one of its subresources. Preemption may be necessary to achieve a form of load balancing in the face of increasing load, or so that clients may acquire multiple resources without concern for having more than their share. Without preemption, the resource management system may have to implement a rather conservative allocation policy or have to provide an unfairly balanced service to users. Preemption should not be used in an overloaded system to reclaim resources only to reallocate them to other similar clients - it could lead to thrashing.

It is desirable to allocate session oriented resources sparsely on a lightly loaded system. However, as the load increases, it may be necessary to contract the set of machines running session resources by assigning several per machine. It is also desirable for clients to be able to acquire as many transaction resources as they need (e.g. for compiling multiple modules), without excluding other clients from having any. Because of the bursty nature of transaction computations, this is less likely to be a problem than the contraction of session machines.

Without resource migration, which is very hard in a heterogeneous system and provides little return on effort, one must reclaim resources or wait for them to finish their task. Reclaiming session oriented resources is feasible if they are allocated directly to users (often the case) who can be expected to execute commands to save the necessary state before reclamation. Reclaiming other session resources or transaction resources is less desirable as the resource and/or program controlling its use must cope with the preemption.

Mechanisms available for notification of preemption were discussed in section 7.5. A simple policy would be to begin the preemption, with five minutes notification, of a "lightly loaded" session resource if it prevented a request for a transaction resource from succeeding. If client hogging of transaction resources proved a problem, clients could be restricted to one or two resources with no preemption, but be allowed more on the understanding they could be preempted with 30 seconds notice.

Miscellaneous Extensions

In certain situations, it is known in advance that a resource will be needed at a certain time for a certain period. Accommodating these situations would require *scheduled allocation* on the part of Resource Manager. With standard "sign up" scheduling, a particular resource instance is reserved for the specified time and no other allocation or reservation which would overlap it will be made. Because the supply of resources varies dynamically and because there may be several ways of providing a resource, such an approach would be unnecessarily restrictive in the Cambridge system. An alternative would be to allow a resource under scheduled reservation to be allocated if there was an alternate way of satisfying the scheduled resource requirement. However, an optimal allocation could require that a number of resource schedules be shuffled. It is unclear how useful a scheduled allocation facility would be.

Accounting, both for the purpose of charging clients for resources used and for limiting allocation of resources has not been considered.

9. Protection and Authentication

Protection must be provided for two parties. Clients requesting resources must be ensured that the resources they receive are authentic, that is, that the resources match their descriptions. The actual resources, or their providers, must be ensured that they are allocated only to clients who have the right to use them.

9.1. Inherent Protection

If resources are in separate physical or virtual machines, and can be accessed solely via public interfaces, a resource can ensure that its internal data is accessed only through routines it provides. Such a programming discipline is certainly possible in a centralized system and its use is becoming far more widespread. However, in a distributed system it is enforced by the fact that components are separate: resources can only communicate via message or procedural requests to interfaces. A service may require that such requests be authenticated and can check that the authenticated client has certain privileges before granting the request.

Because resources are instantiated by the resource management system and subsequently allocated to clients, the services can refine themselves before permitting client access. For example, the Tripos operating system, which is not memory protected, is initially loaded with full access to the root of its filing system. However, before it will accept any user commands or execute any user code it reduces this access as appropriate for the user, deleting the more privileged root access from its memory [Knight 82].

Another advantage of having servers in separate machines in a distributed system, and one of the arguments for the *small server* approach taken in the Cambridge Distributed System, is that data and code space in memory for different services are completely separate. Corruptions in one service cannot directly affect a different service. [Needham 82]

9.2. Protection in the Cambridge Distributed System

Uidsets and Privileges

Any object in the Cambridge system may be named by a 64 bit *unique identifier (UID).* For example, each user, service, resource class, or resource attribute has a permanently assigned unique identifier or PUID. When an object is *active*, it has an associated *uidset* registered in the Active Object Table service. This uidset includes the name of the object (PUID), the name of the authenticating agent (AUTY), an access key (TUID), and a control key (TPUID). The access key is included in requests from the object to clients. The control key is used to manipulate the uidset in the AOT: to extend its timeout, to revoke it, and to enhance it with privileges. The access and control keys are valid only for the duration of the particular activation of the object.

For example, user GSM with PUID *FF02894DCD4E2DCF* will have been authenticated for a session on the Cambridge system by giving his name and password to the User Authentication Service. Thus, his uidset will contain the PUID of GSM as the name of the uidset, the PUID of the User Authentication Service as the authenticator, and newly created access and control keys.

Authenticated requests to clients include the name, authenticator, and access key UIDs from the uidset of the requesting object. (This triple is also referred to as a uidset.) The client may authentically determine the identity of the object by checking the uidset with the AOT for validity. An alternative use of the uidset is as a capability to access the object, in which case it is passed from client to client and eventually back to the object with an access request.

The *Privilege Manager*, another service related to AOT, keeps *access lists* associating privileges (named by UIDs) with object name, authenticator pairs. Any authentic object which is allowed a privilege may ask the privilege manager to enhance its current uidset in the AOT with that privilege. Subsequently, clients may approach the Privilege Manager to determine whether a given uidset has a particular privilege associated with it.

Limitations

There are several limitations with these mechanisms as they are currently provided. Some are due to the basic philosophy and others only to the implementation.

- For an object to prove that it is authentic to a service, it must pass that service its PUID, TUID, and AUTY for the service to check with AOT. However, once the service has that triple, it can then authentically claim to anyone else that it is the original object.
- It is not possible for a user who is allowed a particular privilege to endow that privilege to any uidset other than its own. Thus, a user cannot pass on a privilege without giving away his uidset.
- Though users can enhance uidsets with privileges before passing them on, they cannot refine the associated privileges, thus precluding operation with minimal privilege.

Though these limitations are unfortunate, the available mechanisms do provide a significant amount of protection if certain clients can be trusted.[1] (In any system there must be trust at some level, even if it is in the independent party which verified that the code obeys its specification.) Because this is the most widely used method of protection at Cambridge, and because my interest was in building a resource management system rather than a protection system, the AOT and its related services were accepted as is. A different approach was taken by Yudkin who preferred to emphasize protection mechanisms in his work on resource management [Yudkin83].

9.3. Uidsets for Requests to RM

Most requests to RM from clients must be accompanied by uidsets from which RM can validate that the client is ***bona fide*** and possesses certain privileges. Uidsets must also be verified so that user information for requests can be recorded authentically. This information is intended for accounting use and for listing use of resources to users.

RM maintains ***privilege lists*** indicating which privileges must be possessed by clients making particular requests. For example, there is a list for RM

[1]A full capability mechanism as suggested in HYDRA [Wulf81] might provide more flexible facilities but its enforcement would require a hardware capability unit in the network front end of each machine.

system manager functions indicating privileges required for statistics retrieval and clearing, manual alteration of resource or action entries, changes to the Repository or Action Catalogue, etc.

Entering Resources and Actions

Clients requesting resources of Resource Manager must be able to trust the authenticity of the resource they are allocated.[2] Such authenticity is guaranteed ultimately by system managers (people). This guarantee may propagate through programs which manufacture the resources to RM which can then pass this guarantee to its clients.

Resource Manager maintains this assurance by ensuring that any client which gives it a resource, or an indication of how to obtain one, has the privilege to do so based upon the attributes of the resource. There is a privilege list for entering resources into the Repository associating attributes with the privileges required to enter resources possessing them. Thus RM can accept, for example, Tripos instances only from those clients whom it trusts to provide authentic Tripos instances. There is a similar privilege list for entering and deleting actions for resources with particular attributes.

Most attributes are public: the semantics associated with the 64 bit UIDs are known. Programmers may write these UIDs into their code. It may be the case that some clients wish to offer resources through Resource Manager, possibly to components of the same distributed task, which are not publically known or available to other clients. In this case, random UIDs generated privately may be used to provide protection.

Requesting Resources

Resource Manager keeps two privilege lists indicating privileges which clients must possess in order to use or to be allocated resources with particular attributes. The distinction between privilege for allocation and privilege to use is necessary because a user may be allowed to use a resource, such as a

[2]Alternatively, the client and resource could mutually authenticate each other, though this is a time consuming process. [Girling 83]

Tripos Filing Machine, by virtue of the fact that it supports a resource which he may be allocated, such as a Filing Machine Session, though he should not be allocated the supporting resource, the entire Filing Machine, directly.

For example, people outside of the Laboratory will not be allocated a Tripos because a client requesting a resource with attribute ***Tripos*** must have ***lab privilege***. As another example, the 68000 named Carver belongs to the Mayflower group and has attribute ***Mayflower machine***. It is registered in the ***to use*** privilege list that a client must have ***Mayflower privilege*** to use machines with attribute ***Mayflower machine***. Thus, the machine Carver is capable of supporting a Tripos by virtue of the fact that it is a 68000, but such a Tripos would not be allocated to a non Mayflower user.

9.4. Resource Authentication Information

At Cambridge, addressing and authentication are considered to be separate. Knowledge of the address of a service does not necessarily imply the right to use it. The addresses for static services are freely available from the Name Server, and the subaddresses (port and function numbers) for dynamic services generally remain constant across all instantiations. For requests which must be authenticated, the client must present either its uidset or the uidset or access token[3] for the service. Use by the service of the client uidset is similar to its use by RM, as discussed in the previous section. Use of the service or resource uidset or token is as a sort of capability. The resource trusts that its uidset or token has been returned in a request, having passed only among services allowed to access it.

Authentication and Construction

There are three considerations of authentication during resource construction: authentication of the request to the fabricator or server, provision of authentication information in using the required subresources, and creation of authentication information for the resulting resource. The request includes

[3]An access token is a 64 bit value which may be appropriate for authentication if the full generality and expense of a uidset is not required (e.g. for resources which are created, used, and discarded quickly), or if the resource does not want its uidset passed to the client.

the uidset or access token for the fabricator/server or the uidset of RM, depending on the method in which the fab_serv prefers to operate. RM decides what authentication information to include based on the Repository entry for the fab_serv.

The second concern of authentication is when a fabricator is combining required subresources. As it approaches a resource (e.g. to load it with code or to tell it to cooperate with another resource) the fabricator must provide authentication information in a manner similar to RM's providing it to the fabricator. If the uidset or token for the resource is expected, this will have been passed to the fabricator in the initial request, otherwise the fabricator should include its own uidset.

Third, authentication information might have to be created for the new resource, depending upon which approach it chooses for authentication. This is static for the resource class and will be known by the fabricator or server. If the resource needs to communicate authentically with other network services, then it will need a uidset. As the fab_serv is the component that must guarantee the resource to be authentic (it understands how the resource is constructed), it is the one which must create the uidset. If the resource expects to restrict access to itself with a token, then the fab_serv must generate this token. The newly created authentication information must be made available to the resource, as well as returned to RM.

Capabilities for Debugging

Special attention must be given to debugging because:

- to debug a resource at a particular level, debugging facilities provided by lower levels may have to be used;
- when debugging a resource, proprietary information associated with supported layers must be protected; and
- it may not be known at the time that a service is requested that access to its debugging interface will be required.

When debugging, it is preferable to use facilities which understand the abstraction being debugged. For example, a mail server should be debugged utilizing the facilities provided by the mail server which understand the structure of the message queues and are able to inhibit and enable queue daemons. If there is sufficient integrity to provide this level of service, then authorized access can be enforced by the service itself in the same way that it enforces access to other privileged commands. This does not involve the resource management system.

Debugging a Supported Layer

If the layer being debugged has become too corrupt or it does not provide sufficient debugging facilities, it may be necessary to use the debugging facilities of the supporting layer.[4] For example, if an operating system has become corrupt it may be necessary to work at the bit and byte level by accessing the machine's front end. The resource management system becomes involved because, understanding the structure of the allocated resource, it knows the address of the supporting resource. This subresource interface must ensure that the client accesses only the data and process space of the dependant applications for which the client can provide authentication information.

Debugging a Supporting Layer

When debugging a layer that supports other services, access to possibly proprietary information contained in the data space of the supported resources would be available through the debugging interface. Therefore, the client must obtain permission from (i.e. authentication information for) all supported services, not only at the next level but at any higher levels.[5] For example, the designer or maintainer of a database system supporting a payroll package should not be allowed to debug the database system without permission from the manager of the payroll package.

[4]If each layer of a service provides a protected environment on which a supporting layer can run, then errors of the supported layer cannot pass through this firewall to corrupt the supporting layer. Though some interfaces still do not guard against requests which can corrupt their internal state, such lack of caution is generally regarded as bad programming practice (or necessitated by performance constraints or lack of memory management hardware and language type safety) and will not be of concern here.

[5]A domain approach as used in CAP [CAP 79] could be used within a layer to allow cross debugging of domains, but could not be used to meet the above requirements because the debugging facilities of one layer actually do have a hierarchical relation to the next.

A designer or maintainer of a particular type of service will have a privilege indicating the right to debug it. The resource management system, because it understands the structure of resources, can accept from such a privileged client a request to debug a resource of that type, along with the authentication information for all supported resources, and return the information for the desired debugging interface. A separate privilege list would be required to determine privileges required to debug resources with particular attributes.

Current Implementation

In the current implementation, the authentication information for a resource also provides access to its debugging interface. If a client has been allocated a resource he has the right to debug it, and will be given the authentication information to access any supporting resources. Some allowance of this sort was necessary because several of our current systems do not provide protected environments for supported layers. If a client has the privilege to debug a particular type of resource, he will be given access to that resource's debugging facilities regardless of what other services it supports. This was for simplicity.

10. Miscellany

This chapter discusses several miscellaneous aspects of resource management: the preloading of resources, the bootstrapping of the Cambridge Distributed System, and distribution.

10.1. Preloading

As part of Resource Manager's goal to respond to requests promptly, it will keep a stock of free resources ready for immediate allocation by obtaining resources from servers or fabricators ahead of time. This process is termed *preloading*.

Preloading was found to be necessary because of the long time taken to obtain or construct resources, and if they were services, for them to subsequently initialize. In the case of loading an operating system into a machine, the actual time for loading code can be saved, as well as can the time for the operating system to perform non user-specific initialization, such as setting up its file system and obtaining the current time from the network clock. At Cambridge, in the case of Tripos, these times initially combined to represent 12 to 25 seconds of idle user time.

10.1.1. Prediction

Because RM does not have unlimited resources from which to construct spare, higher level ones, it must prepare those resources which it ***predicts*** will be requested. This prediction is based currently on a record of past requests, though it could include such factors as current time, day of week, etc.

Exact prediction of which resources will be requested next would depend on knowledge of client behavior, which is generally not available. Therefore, only an approximation can be attempted. Whether a requested resource has been preloaded does not affect whether the request will succeed: it only alters the response time. Given these considerations, the decision of what to load would not seem to merit expensive analysis of the input data.

Using information in the request records, the Repository, and the Action Catalogue, it might be possible to attempt some optimal combination of available free resources (e.g. the one giving the most preloaded resources). Such effort does not seem worthwhile, as probably only a few resources will become free at a time and interactions with fabricators or servers may fail.

Data and Analysis

Currently, the preloading module keeps a record of the past 40 satisfied resource requests with the purpose of determining which resources were most recently requested and in what relative quantities. Only requests which were satisfied are recorded so that client retries on failure do not distort the figures. Requests specifying that private resources or actions be used should not be recorded, but requests specifying optional use of these resources or actions should be.

The current intention is to attempt to retain particular free resources in the same ratio as they have been requested in the recent past. In analyzing the request records, equal weighting is given to all requests, though it might be more reasonable to use linear weighting to give priority to the more recent requests. The expense of including some (exponential) decay of requests over time seems unnecessary.

The result of analysis is a list of which resources should be preloaded and in what quantities. It reflects the ratio of resources in the request records, less any free resources already available.

10.1.2. Construction

In addition to being notified of requirements given in successful resource requests, the *preloading* module is notified when a resource becomes free or when a new resource or action is entered, as these events may make it possible to preload another resource.

When to Run

It is not feasible for the preload process to run as a background (low priority) job which is suspended when a client process is started because it may hold resource reservations preventing the client request from succeeding. Ideally, the preload process should request resources at the earliest time when such a request will complete before a user request intercedes. In the current implementation, after being notified of one of the above events RM waits until system activity has settled (i.e. there are no outstanding resource requests), and then waits an additional nominal time (30 seconds) as resource requests often come in groups. At this point it analyzes the data in the request records to predict which resource, of which it does not already have sufficient free instances, will next be requested.

Requests

Once the resources which should be preloaded are determined, RM successively places requests through the standard channels for them. It will indicate that the status of the resulting resources will be ***free***. If a request for a resource of a particular type fails, no more requests for that type will be attempted. If a client request for a resource is received, the preloading activity is terminated as soon as reasonable.

This simple strategy for analysis of preloading data has proved satisfactory; the vast majority of requests which can be satisfied are satisfied from the preloaded pool. Unfortunately, the current situation is not very demanding because almost all requests are for one of a handful of resources (one of the Tripos variants or Mayflower) [section 12.1]. Whether it will perform as well as the situation becomes more demanding remains to be seen.

10.2. Bootstrapping the Cambridge Distributed System

This section presents a somewhat idealized version of how the Cambridge Distributed System could be brought into existence from a complete shutdown. The concentration is on the relation of bootstrapping to resource management. Such presentations are inevitably specific to the system being considered.

10.2.1. The Difficulty

There are two problems:

- services must be brought up in an order consistent with their critical interdependence; and
- services must be instantiated authentically.

A minimal subset of services, and the approximate order in which the components must be brought up, is: Name Server, File Server, Boot Server, AOT, Ancilla, bridges/gateways, RM, Machine Server, Aliveness Server, Event Notification Server, and terminal concentrator or user workstations. The world initialization routine in the Boot Server handles most of this.

10.2.2. Services and Loading

Some servers self initialize, bootstrapping from PROM or their own discs. The number of these services is kept to a minimum to avoid unnecessarily complicating the code (particularly of small servers) and because it ties servers to particular machines. Most servers are loaded by other components of the system. There must be a special mechanism for authenticating such external loading until the authentication server is instantiated.

The Name Server and the File Servers

The Name Server is absolutely critical and must initialize first. Its program and a few critical name table entries are in PROM, which it reads on power up. The Name Server eventually reads its full static name table from a File Server.

The File Servers bootstrap off of their own discs.

Authenticated Loading

In general, requests from fabricators to load machines will include a uidset which the machine's front end or PROM loading program will check with the AOT for validity and with the PrivMan for an appropriate privilege. Such uidsets will generally be from the Boot Server or one of the Ancillae.

When a client receives an address from the Name Server or Resource Manager, it trusts that the code running at that address implements the service or resource it requested. For the statically assigned services, authenticity is ensured by the machines themselves at load time by accepting load requests only from services they trust to load the correct code. These machines might check for specific privileges such as ***load ancilla privilege*** or ***load spooler privilege***, or they might be less paranoid and check for a ***load static server privilege***. For the dynamically assigned services, authenticity is ensured by Resource Manager as discussed in section 9.3.

AOT and Related Services

The problem at the start of day with this scheme for authenticated loading is that the authentication services don't exist. To overcome this, the requirement for checking a uidset must be satisfied in some alternate manner until the AOT can be booted. When a load request is sent to AOT a "system password" replaces the TUID in the accompanying uidset. (A full uidset is used for consistency with normal load requests.) The PROM for the AOT Z80 indicates that when a load request arrives, rather than checking the given uidset with the AOT, the TUID in the request should be compared with the UID in PROM. If they match, the request will be accepted.

The Boot Server

It is the responsibility of the Boot Server to bring the Cambridge Distributed System into existence. Most of the special cases of loading code for initialization are done by the Boot Server, either directly or through requests to other servers which it has already booted. The most notable exceptions to this are the Name Server, which the Boot Server must access to find where to load static services, and the File Server, from which the AOT must recover its stable state before machines can be authentically loaded.

The uidset for the Boot Server may be persistent, that is have a timeout which will survive distributed system failures. This would allow it to have a valid uidset whenever it initializes, and thus be able immediately to make authenticated requests to other services. (The TUID of this uidset would be the system password for AOT mentioned earlier.) Because it must pass a uidset to any machine it wishes to load and because it is unwise to present a persistent uidset to any service but the AOT, one of the first things the Boot Server should do is to use this uidset to have the AOT create it a temporary uidset which it can more safely distribute to machines.

The Boot Server must be implemented as a stand alone system. There should be a backup Boot Server in case the primary one fails. There are no dependencies on where the Boot Server resides; it must only possess the persistent uidset mentioned above. Server code (including that of the AOT) may reside on the File Server where it can be accessed by any of the candidate Boot Servers.

10.2.3. Initialization

This section briefly discusses the sequence of events which occur when power returns to the distributed system after a complete shutdown.

Static Services

The Name Server boots off its PROM and attempts to read its full name table from the File Server. It backs off if this fails, retrying later. The File Servers have a bootstrap program in PROM which loads the fileserver code from disc. Once these have initialized, the Name Server request for its full name table will succeed. The Boot Server runs on a stand alone system (currently a PDP11 running RSX or a VAX running Unix) and will initialize according to the host operating system. It provides the Z80 loader, 6809 loader, and world initialization routine.

The world initialization routine in the Boot Server looks up the address of the AOT in the Name Server and attempts to obtain a temporary uidset using its persistent one. If this succeeds, then the AOT is functioning and it is probably the case that the Boot Server just crashed and rebooted but that the distributed system world is operating. Otherwise, the Boot Server should now authentically load several servers. However, before authenticated communication among network components can take place, the AOT must be booted.

The Boot Server retrieves the AOT code from the File Server and sends a load request to the AOT. When the AOT Z80 receives the request, it compares the accompanying TUID with the TUID in its PROM and, if they match, accepts the code and begins execution. Once the AOT is loaded, it will read the values of its internal tables from the File Server. The Boot Server now obtains a temporary uidset for itself from the AOT, using its persistent one for authentication.

Next, the world initialization routine needs to load the 68000 Ancilla (a Z80) before it can load RM. It looks up the address of Ancilla-68000 in the Name Server and has the Z80 loader (part of the Boot Server) load it, this time using the newly acquired temporary uidset for the Boot Server for authentication. The Z80 retrieves the Ancilla bootfile from the File Servers, creates and links in a uidset for Ancilla-68000, and sends the code to the Ancilla Z80. The Ancilla Z80 PROM code will verify the uidset from the Boot Server and then check that it has an appropriate privilege. If these checks succeed, then the load request will be accepted and the Ancilla will come into life.

The address for Resource Manager is looked up in the Name Server, and a similar procedure to the one for booting Ancilla is followed to boot RM's 6809 network front end. Next, the Resource Manager itself is loaded by sending an authenticated load request to the 68000 Ancilla, which obtains the RM code from the File Servers, creates and links in a uidset for RM, and loads this into RM by sending an authenticated load request to its 6809 front end. When RM initializes, it will read its stable state from the File Server. Note that several services have been brought into existence about which RM doesn't know but should know. These include the Z80 and 6809 loaders, and the 68000 Ancilla. The world initialization routine makes entries into RM for these fabricators.

There may be several more services which must be brought into existence for the world to be in a usable state. The terminal concentrators are likely candidates. The world initialization routine sends requests to RM for each of these.

Dynamically Instantiated Services

Now that RM is available, any other servers which are pointed to by the Name Server may be brought up by requests from the world initialization routine directly to RM. These services include Clock/Logger, Mercator, PostMan, and Spooler. All of these actually could be brought up (on a dynamically allocated machine) only when needed if Ring clients approached RM for these resources rather than going directly to the Name Server. However, retrofitting such a procedure to all the Ring software is infeasible.

Any other services for which clients approach RM, such as Mayflower or Tripos, will be instantiated and allocated as required.

10.2.4. Failures

Hardware failure of dynamically allocated machines generally does not require immediate intervention as the supported service will be reinstantiated on a different machine when next requested. However, hardware failure of a static server such as the Name Server must be fixed. For processor bank machines, this may generally be done by replacing CPU, memory, or Ring station boards, or the coding plug (giving the station number) may be swapped to a working machine. There should also be spare PROMs for the Name Server as well as for the static services, such as AOT.

The Aliveness Server and the Event Notification Server (discussed in sections 7.2 and 7.3) are used heavily to detect failed services, and reboot them if appropriate. Because RM starts a dead man's handle for all resources entered into its Repository, all running code (except the Name Server and File Servers) will shake dead men's handles and their failure can be detected. The Aliveness Server shakes a dead man's handle with the world initialization routine.

The Resource Manager will have registered an RPC with the Event Notification Server to be made for each resource which fails (as determined by the Aliveness Server). If this call is made, RM simply deletes the entry for the resource from its tables. For a dynamically instantiated resource, nothing else need be done as RM will reinstantiate it if another request for it arrives. For a static resource (Ancillae, Pack Server, AOT, terminal concentrators, etc.) the world initialization routine will have registered an interest in the resource's failure, giving a resource request to be sent to RM to reinstantiate the resource should such a failure occur.

10.3. Distribution

Resource Manager is a very important component of the distributed system. Thus, provision must be made for its robustness, including the ability to survive crashes. Distribution by replication and state saving were the two alternatives considered.

The design and implementation for distribution were only partially completed. Other aspects of Resource Manager were felt to be more important, particularly in view of the high reliability of machines and the network at Cambridge, and of how hard it is to get distribution right. State saving, in which copies of the Repository and Action Catalogue are kept up to date on network fileservers, was implemented to allow recovery of state after a crash. Although this is not as general as distribution, it is much simpler. However, distribution is felt to be sufficiently important to merit a brief description of the plans which were being implemented.

There would be several RMs active at once, each with a complete copy of the Resource Repository and Action Catalogue. Any request could be directed to any RM and it would be satisfied by that RM, i.e. the client would not be redirected. The RMs would reconfigure as necessary to compensate for lost RMs and to accommodate new ones. Network partition could result in isolated groups of RMs and resources; these groups would recombine when the network is restored.

10.3.1. Controlling RMs

Each resource would be assigned a *controlling* RM, which would be responsible for synchronizing any transactions involving the resource. Such transactions would include entering and removing resources, allocating them to a client or for use in fabrication, and changing a resource's controller. Any time an RM wished to undertake such a transaction, it would first contact the resource's controller. Before the controller could acknowledge completion of the transaction it must inform n other RMs of the state change, where n is a function of the reliability of RM and the network, and of the required reliability of the system.[1] The controlling RM would try to notify the remaining RMs of the state change after acknowledging the transaction. If any RM attempted a transaction involving the resource before it was notified of the state change, it would be negatively acknowledged by the controlling RM.

Initially, a resource's controller would be the RM which created it or at which it was entered. A RM wishing controllership of a resource would propose the transfer to the resource's controller. All such proposals would probably be accepted, though a suspicious or protective RM need not do so. The RMs would attempt to keep a balanced resource distribution by requesting resources from overendowed RMs if they had less than their share of resources. When a new RM comes into existence it would request the Repository and Catalogue state from the nearest RM and then proceed to request control of its share of resources as described above.

10.3.2. Distribution and Reservations

Recall that when satisfying a resource request a blueprint containing nodes for all the resources involved in the transaction is assembled, that reservations are made for each existing resource to be used, and that only when a complete plan is in hand is any construction begun. The choice of what and when to communicate among RMs when satisfying a resource request is important. Too much communication would cause unnecessary delays. There is little point in conversing about reservations as there seldom will be simultaneous requests to different RMs requiring the same resource. The resource's controller will ensure mutual exclusion if this does happen.

Thus, communication would be undertaken only when the blueprint had been assembled and construction was to begin. It would be too expensive to

[1]Increasing n could compensate for a failure prone system. However, it would be unnecessary to make the resource management system more robust than the resources which it manages or the paths through which they are accessed. In a local area network in which the hardware and support systems could be made fairly reliable, n is likely to be 1.

communicate individually with the controller of each resource or subresource involved in the request, especially as each controller would then propagate the state change of its resource to all other controllers. It would also be complex to reverse all the allocations which had succeeded if one of them failed.

As a (necessary) optimization, allocation of all of the resources in a blueprint would be handled as a single transaction coordinated by the originating RM. Information about the required resources would be placed in a request which would be sent to the controllers of all the involved resources, each of which would confirm or deny the allocation of its resources. If any allocation was denied, the originating RM would send a similar block request to reverse the successful allocations. At least n RMs would be notified synchronously by the originating RM; the remaining would be notified asynchronously.

10.3.3. Failure and Recovery

The failure of a Resource Manager would be more complex because resources would be left without a controller. In this case, an election would be held according to a slightly modified version of the Bully Election Algorithm [Garcia-Molina82] to determine who would inherit control of the orphan resources. This algorithm is so named because the node with the highest node identification number will force election of itself. When a Resource Manager determined that another RM was down, it would call for an election with itself as coordinator of the orphan resources. Responding to a two phase commit protocol request, RMs with lower identification numbers would concede the election while those with higher ones would refuse it and then start their own.

If the network partitioned there would be isolated groups of RMs and resources. After elections, there would be one RM in each group which considered itself the controller of a particular resource. However, only the controller and clients in the same partition as the resource would be able physically to access it. It would appear to others that the resource had died. When the network rejoined, RMs would merge their state with that of ones which were newly accessible, resolving controller conflicts by chosing the RM with the higher identification number.

11. Other Approaches to Resource Management in Distributed Systems

This chapter was not entitled "Related Work" because, while all the work described here deals with resource management in distributed systems, the basic set of premises is generally rather different from those at Cambridge. This chapter also includes a discussion of a major alternative I pursued and of that pursued by a colleague of mine, Mark Yudkin.

11.1. File and Process Resources

A number of distributed systems [Locus83, Jade83, NSW77, RSEXEC73] consider only files and processes as the resources available among components of the system. They provide different degrees of support for remote access to files and for execution of processes on remote hosts. Most of these systems are homogeneous in that each node runs the same system, either directly or as a guest layer (though the underlying machines may be different). Such homogeneity allows simpler and more intimate resource sharing at the cost of excluding the use via the same mechanisms of resources available from other systems. As such, the requirements for resource management in these systems are very different from those in the Cambridge system.

11.2. Object Based Systems

Several distributed systems [Eden83, Zahorjan85, Argus84, StarOS79, Medusa80] have provided support for objects which are similar to resources in the Cambridge sense. Each of these systems depends on a homogeneous environment.

Eden is a message based, object oriented system in which objects, as they are relatively expensive (at least three processes each), are reserved for reasonably high level abstractions (e.g. mailboxes). The location of an object on the network is transparent to clients, as in the Cambridge system, but it may also change over time - an important feature as Eden objects are generally more long lived than Cambridge resources.

An object of a particular type is created at a specified node, which defaults to that of the type's manager, by sending a "create" message to the appropriate type manager object. (Creators of type manager objects must say explicitly where they are to be located.) During invocation, an existing object is located by querying its type manager, whose location is included in the capability for the object. Type managers are guaranteed to know the current locations of all objects of their type.

In *Argus* a distributed program is composed of ***guardians*** (e.g. mailers, print spoolers) which encapsulate and control access to resources. They can be created dynamically, with the programmer specifying the node at which the guardian is to reside. A distributed ***catalog*** is proposed to allow late binding of guardian names to one or more addresses.

The specification of a guardian is by a single name, which contrasts with multiple attributes for a resource in the Cambridge resource management system. Much of the need for monitoring, reclamation, and dynamic reinstantiation of resources in the Cambridge system is subsumed in guardians by the use of atomic actions and automatic recovery to provide resilence. A consistent mechanism for both accessing existing guardians and having new ones created dynamically, as there is for resources in the Cambridge system, could be provided by functionally extending the proposed catalog.

Task forces in *StarOS* and *Medusa* (message based, object oriented systems for the Cm* multi-microprocessor) are collections of concurrently executing processes that cooperate to provide a single logical resource, such as a concurrent compiler, an operating system, or a filing system. Their use bears similarities to the use of Cambridge resources, with ***descriptor lists*** in Medusa for providing addresses of remote utilities. However, the placement of processes and objects within a task force is more important than in the loosely coupled Cambridge Distributed System.

11.3. Personal Computers

The use of idle machines in a network of personal computers as available resources has been proposed by Shoch and Hupp [Worms82] and Dannenberg [Dannenberg82]. In both of these systems, attention has been paid to the autonomy of personal computers. Users must actively offer their machines for use, and may reclaim them at any time.

In Shoch and Hupp's work, a ***worm*** program is composed of an optimal number of ***segments***, each running on a different machine. To obtain a new segment, a worm broadcasts on a well known socket, and awaits a reply from a consenting host upon whom it will then load the new segment. A worm must retain sufficient distributed state (use of local discs is not allowed) to recover from a segment's being aborted at any time. This, along with the requirement that a worm propagate itself, makes worms a rather limited class of application.

Dannenberg has developed, in conjunction with the Spice project at CMU, an operating system component known as the ***Butler***. It will accept a specification for a remote resource and a list of potential hosts to be tried in turn. The specification includes the resource name (a string), a list of low level subresources which must be available (an arbitrary collection of name, value pairs), such as memory, cpu time, and disc pages, and a list of servers to whom access will be required (strings). These are similar to the class, attributes, and required subresources of an action in the Cambridge RM.

As a remote host, the Butler awaits an incoming request and consults its policy database to determine whether it can meet the specified requirements. This may involve checking the ***signature*** of the client with the ***Banker*** service to authenticate the client and to determine numerical limits on privileges. The AOT and PrivMan services are used similarly in the Cambridge system. The Butler then monitors the execution of the guest program, notifying the client Butler if the program exceeds its subresource limits or if the host user demands the return of his machine.

In this system, a resource is viewed basically as a command (with requirements for server access and internal resources) running in one of a number of machines suggested by the user. It depends upon a homogeneous environment: single machine type, single operating system, single language.

11.4. The Dynamic Nameserver Alternative

An alternative approach which I pursued involved incorporating management functions into a dynamic nameserver (a server providing name to address mappings which change on a relatively frequent basis). This scheme would have incorporated the static nameserver.

Initially, the dynamic nameserver would know the addresses of only a few other services, such as the authentication server and other dynamic nameservers. As services came into existence, they would notify the dynamic nameserver of any resource they were willing to provide, the address of this resource, and some authentication information. For example, the network front ends to machines might have a simple program in PROM which would contact the dynamic nameserver on power up or reset, giving its machine attributes and the network address on which it was willing to accept code to be loaded (subject to the authentication token which it handed to the dynamic nameserver being returned with the request).

Fabricators such as the Ancilla might enter names for Tripos and Mail, stating that the result would be an address for further indirection. Thus, in response to a request for the address of a Tripos, the client would receive the address of the Ancilla with the indication that it was being redirected. The client would then redirect its request to the Ancilla, and expect back a network address for the target service. In satisfying such a request, the Ancilla might have recursively called upon the dynamic nameserver to obtain the free machine which it loaded to provide the target service.

The resource management functions of the dynamic nameserver would include periodically contacting the registered resources to determine if they were still alive, and removing entries for ones which failed to respond.

Another intended management function was to provide the *quality of service* and *allocation time* factors as described in section 8.3. These factors would be entered into the dynamic nameserver as part of resource registration. For bottom level resources, such as machines, they would be constant. Fabricators and servers should be able to calculate these factors for resources they offer based upon the construction involved and the value of the factors for resources to be used in the construction. The fabricators and servers could periodically check the availability of resources they would use in construction and alter the corresponding entries in the dynamic nameserver. Thus, if there were no machines available on which Tripos could run, the Ancilla could retract its offer to provide a Tripos. If only LSI4s were available, it could reduce the quality of service parameter accordingly. As these factors would only be hints, their accuracy would not be critical. Changes for factors for low level resources would eventually trickle up to the entries for higher level ones.

This scheme was eventually abandoned for several reasons.

- Accommodating multiple level resources required that either all of the servers be present to make entries for resources which they could provide, or lower level ones had to know all the possible resources which could be supported by any resources they provided.
- Fabricators could not ensure they would be able to obtain multiple required subresources for fabrication, particularly if two of the subresources were of the same class. Reservations would have led to a complex thread of execution through multiple machines with a considerable number of network messages being passed sequentially.
- Clients would have had to be prepared to redirect their requests.
- There was difficulty in deciding which attributes of the client request could be satisfied by which levels.
- There was no central repository of information for use in making policy decisions about resource allocations.

11.5. Rochester's Intelligent Gateway

Rochester's Intelligent Gateway (RIG) [Lantz 82, Lantz 80] is a distributed system designed for a collection of heterogeneous machines interconnected by networks of varying characteristics. RIG's emphasis is on inter-machine message passing, remote process management, and a general purpose intelligent user interface with table driven command interpreters. RIG is particularly interesting because it shares some of the premises of the Cambridge Distributed System: a heterogeneous group of machines supporting different operating systems which are integrated to varying degrees, and resources being viewed as higher level objects available among system components.

Though both systems support similar servers (a nameserver, fileservers, printservers, terminal concentrators, a resource/process manager, a logger, and a clock service), in the Cambridge system these are provided as small servers in separate machines but in RIG as server processes on each host. Thus, for a host to be fully integrated into the RIG environment, it not only must support the RIG protocols for communication, but must provide a reasonable complement of these server processes. In the Cambridge system, only the Ring protocols (SSP, BSP, VTP) need be provided for full integration, and the host can then access servers on the Ring in the way a RIG host accesses the server processes which it supports. However, in the Cambridge system a host is dependant on the Ring servers, whereas a RIG host is largely self supporting and can run stand alone.

Processes as Resources

In RIG there are resource or process *profiles* which map keys such as "editor" into the associated code segments to be loaded. The local process manager (one of the RIG server processes) takes such a string and returns the process identifier of a newly created process running the associated code. This is similar to old RM's mapping a name for a service into an Ancilla loadfile, having the associated code loaded, and returning the name of the allocated machine.

The process manager keeps a tree of spawned processes which it uses primarily to avoid orphans. When a process dies, all of its subprocesses (both local and remote) are notified and given a cleanup time, after which they will be killed. Process managers are notified of the death of remote processes by the network servers, whereupon they will notify any processes which have registered interest in the remote process death (generally servers wishing to release allocated resources). This mechanism is similar to that provided by the Aliveness and Event Notification servers in the Cambridge Resource Manager except that in RIG, the interest is contained to one machine, there are a small fixed set of events concerning process activity, and notification is by a signal rather than by executing an arbitrary procedure.

Service Location

Processes willing to provide a service (named by a unique string) send a message indicating this to the nameserver in their host. Subsequently, when the nameserver receives a request for a service, it returns the address of the appropriate registered process. If the requested service is not in the nameserver (i.e. not available on the local host), the local nameserver notifies each network server that it should propagate the request on its associated network. The remote network servers communicate with their respective nameservers to determine which host, if any, provides the service, and the resulting address is returned to the original requestor. If more than one host offers the service, all "bids" but the first are ignored.

Similar function is provided by the Cambridge Resource Manager. Any service may indicate a willingness to provide resources by entering one or more actions, in which case they will be approached for the resource when needed in the same sense as the RIG servers. However, with RM it would also be possible to enter resources directly which would be like precreating processes [provided by servers] in RIG. Additionally, RM has a number of fixed actions for standard resources. By contrast, the RIG equivalent to the fabricator or server must exist and have registered a willingness to provide the resource. It is suggested that the RIG nameserver could act as a canonical server, creating the appropriate server process if none were already available.

11.6. Yudkin's Resource Management System

A colleague of mine, Mark Yudkin, has concurrently developed a resource management system for the Cambridge Distributed System [Yudkin83]. Though starting from the same premises, our emphasis and goals were somewhat different. Authentication and security played a primary role in Yudkin's work, and he considerably augmented the current Ring authentication system to achieve these. By contrast, I concentrated on the description, location, allocation, and construction of resources. Such differences account for part of the difference in resulting approaches, though much of it is simply because we felt different solutions to be appropriate. I will give an overview of Yudkin's system and point out areas where it differs from my approach.

Overview

Yudkin's proposed system consists of two primary types of services: the Manciple and a number of Service Secretaries. The ***Manciple*** accepts resource requests which it reroutes to the appropriate Service Secretaries. A request includes a description of the desired resource and what access rights to it are needed, as well as the client's uidset or an access key. Once the requested resource has been obtained from a Service Secretary, the Manciple uses this authentication information to restrict the access rights in the capability for the resource, and then passes the capability and resource address back to the client. The Manciple determines the allowed rights based upon the client's privileges as registered in the AOT Privilege Manager. The client may then include this capability in his request to the resource, which will contact the Manciple to verify the capability and determine the client to whom the allocation was made and what rights he is allowed.

There is a separate ***Service Secretary*** for each type of resource available in the system, which is responsible for providing instances of that type resource. To satisfy a request, a Service Secretary might allocate the resource from a pool of available instances or contact the Manciple to acquire subresources and combine them into the target resource, possibly with the assistance of ***booters***. The majority of policy for the allocation of resources of a particular type resides in that type's Service Secretary.

A Service Secretary will accept a resource/service request only from the Manciple. The request must include the resource name, attributes, and rights for the desired resource as well as the Secretary's access key which only the Manciple knows. No information about the user is given. The response must include the resulting resource's address, authentication information, and access rights, as well as capabilities for any subresources obtained (as the Manciple retains the graph structure of resources).

When a client finishes with a resource, he will notify the Manciple which will then notify the Service Secretaries of that resource and all subresources. Though no provision is made for detecting the failure of a resource, Yudkin postulates that the Service Secretaries could poll the resources and interact with an Event Notification Server (as I proposed) to notify appropriate parties.

Advantages and Disadvantages

Authentication and Protection

Yudkin's attention to authentication and protection is excellent. He addresses the problem of rights associated with capabilities which can be restricted or enhanced. More importantly, he addresses the problem in the current system of authenticating clients by passing uidsets (thus giving the recipient the ability to prove that he is the client) by using the Manciple as a trusted intermediary. This is a good scheme, though I believe it should be part of the authentication (AOT) services rather than part of the resource management system. As it stands, resources obtained through the resource management system do not have the user's uidset, and thus cannot interact with services accessed outside of the resource management system. For example, Tripos cannot send documents to the printer spooler because the spooler requires the client's uidset. Forseeing this problem, Yudkin has suggested that the user be required to handle such interactions directly.

Separation of Policy - Distribution of Function

The other main advantage of Yudkin's system is that the vast majority of allocation policy for resources has been separated into the Service Secretaries. The Manciple enforces rights restriction based upon privileges possessed by the user, but otherwise it is completely up to the Service Secretaries to decide upon allocation policy. The policy can be tailored to the resource being provided. Unfortunately, Yudkin has failed to provide the Service Secretary with any information about the user for it to incorporate in policy decisions, probably by oversight.

A more serious problem in this area is that the Service Secretary has only local information upon which to base the allocation decision. For example, the policy that only three of the 68000s may be allocated to run student systems, such as the student version of Tripos or the student version of the mail system, cannot be enforced as it requires information about the allocation of several types of resources and each Service Secretary has information about only one type. In my resource management system, the allocation information about all of the resources is available in the Resource Manager in a consistent form, thus facilitating its inclusion in policy decisions (though a small number of policies must contend with all types of resources).

The unavailability of infor[illegible]em is, I believe, its biggest flaw. The deciding factor in distribution seems to have been that policy must be delegated to separate servers, with the consequences that there is no central repository of information for making these decisions and a considerable amount of network communication must be carried out between the Manciple and Service Secretaries.

The result is that it takes 27 seconds to obtain a Tripos resource.[1] Yudkin states that only five seconds of this time is spent loading code. The remainder is spent in the Manciple, in the Service Secretaries, in the communication between them, and in the communication with the Active Object Table. Yudkin suggests that the user "issue the request and do something else during the half minute it takes to satisfy that request". Unfortunately, this half minute could be considerably increased if reservations were used or if more than a simple two layer resource had to be fabricated.

Yudkin comments that preloading could be used to avoid this long allocation time. I disagree. The preloading would save only the five seconds of actual machine loading; because each Service Secretary implements the policy for its resource, the same communication paths between the Manciple and Service Secretaries must be followed once the ultimate user of the resource is known so that this information can be taken into account in the policy decision. Additionally, in his suggestion for preloading, Yudkin does not inform the server or fabricator of the intended status for the resource. A server may not be willing to provide a resource which will only be placed in a free pool, as it may tie up internal resources of the server.

[1]By comparison, the times for obtaining a Tripos from RM II via RPC are:
- 7.8 seconds building resource from scratch (comparable to Yudkin's task - treating RM I as a machine server);
- 5.1 seconds with 68000 already in Repository; and
- < 0.1 seconds with Tripos preloaded onto a 68000.

In his basic system, Yudkin has ignored any sort of reservations with the result that time may be spent fabricating a subresource only to find that a co-required subresource is unavailable. The reservation system which is eventually suggested would take place across Service Secretaries, increasing the required network communication.

Service Secretaries

One of my goals was to explore the commonality of resource allocation policies and provide a general mechanism for allocation at the expense of individually tailored ones. This makes it is very easy to handle new resources. Network servers willing to provide them simply approach Resource Manager with the resources or with an offer to provide them on demand. Actions can be entered describing how to construct resources from existing subresources. Yudkin opted for the more flexible approach of individual policies for each type of resource available on the network, with the resulting cost that offering a new type of resource requires implementing a Service Secretary for it.

Additionally, for Yudkin's scheme, a Service Secretary must be in existence for each type of service on the network. To ensure this, Service Secretaries are treated specially by the Manciple which interacts directly with booting services to have them constructed. Yudkin has explicitly said that the interfaces to booting services may be varied, with the result that the Manciple must understand a number of booting service protocols for the different types of machines on which the Service Secretaries are loaded.

Yudkin's system provides no way for providers of services to offer them to the remainder of the community, although it could be done by extension of the Manciple and Service Secretary interfaces.

Attribute Trees

For description of resources, Yudkin requires attribute trees reflecting the resource structure. He admits that, due partly to the difficulty in passing this information between the Manciple and Service Secretaries, severe restrictions have been imposed in the implementation. I chose to flatten the attribute space so that users did not have to know about the structure of resources, with the resulting restriction that some desired structure of resources could not be specified. For example, it is impossible to request a particular compiler server and specify that the front end is to run on a 68000 and the back end on an LSI4 if there are actions for providing the back and front ends on both 68000s and LSI4s. Yudkin potentially does not have this restriction.

12. Use of Resource Manager

This chapter explains how I see the user interacting with Resource Manager and the processor bank. Much of what is described doesn't involve Resource Manager directly, but provides a setting in which it may be used. Some of this work is already available, some is currently underway, either through my own initiative or through Project Mayflower or the Programming Environments Research Group, and some is only speculation. I anticipate that RM will continue to evolve as more and more of this work is completed.

12.1. Conventional Processor Bank Use

Providing Operating System Instances

The processor bank is largely used, as it has been historically, to provide connections to dynamically instantiated operating systems. There are several variant configurations for Tripos, both for LSI4s and 68000s. There is also a configuration for Mayflower for 68000s. With the new MicroVax processor bank (currently three MicroVax I's; 12 MicroVax II's to arrive this summer), Ultrix should soon be added to this list of available operating systems, as will VMS in the longer term.

There is a *Machine Server* which keeps a list of the existing machines (LSI4 and 68000 based systems) with notations of which have been allocated to RM or withdrawn for maintenance. There are resource actions in RM indicating the class and attributes of machines which may be obtained from the Machine Server. The Machine Server registers an interest in the expiry of any of its allocated machines in the Event Notification Server. Upon such notification it has the appropriate Ancilla reset the machine and then marks it as free. The Machine Server is a Mayflower server written in CLU.

Instantiating Static and Dynamic Servers

The Z80 small servers are still reloaded by the Boot Server after a crash or when the Z80 reset button is pushed. Either event generally causes the Z80 to branch to a fixed address in PROM memory which results in the Boot Server being prodded. It would be possible for RM to instantiate these services, based upon actions in its Catalogue, by using the Boot Server as a fabricator for Z80 services. As it would be infeasible to change all clients to request service addresses from RM rather than the Name Server, these services should remain static (using machine names as attributes to determine location). The advantage, aside from the elegance of consistency, would be that the Z80s which are not statically assigned could be allocated dynamically by RM.

There are four major static servers running on processor bank machines from the Machine Server: Resource Manager itself, the Canon Print Server, the Tripos Filing Machine, and the Tripos Mail Server. The Print Server is static by virtue of special hardware. The attribute *Canon* is associated with the Print Server machine and is required by the Print Server resource action.

The Filing Machine and Mail Server could be instantiated dynamically (i.e. in any appropriate free machine) except that clients approach the Name Server for their addresses. For these services the machine names are attributes of the machines, and are required by the Filing Machine and Mail Server actions. Currently, when they crash they are reloaded through RM by system administrators. With minor modification they could, upon initialization, register an interest in their own death with the Event Notification Server giving a request to be sent to RM to reinstantiate them when such an event occurred.

Other Dynamic Services

There are several dynamic services available in addition to the operating systems mentioned above. These include the Rainbow high resolution color display system, a Wirewrap system requested by the Pointing Machine controller, a Garbage Collector requested by the File Servers, an Accounting Daemon requested by the Tripos Filing Machine, and a Mail Daemon requested by the Postman server.

12.2. Mayflower

Project Mayflower is developing an environment to facilitate distributed programming. It is based on the CLU language which has been extended with concurrency and synchronization primitives, as well as with remote procedure calls for cross machine communication. The Resource Manager is used for the naming, location, and construction of network resources. Present research interests include: a distributed compilation system, debugging of distributed programs, atomicity for abstract objects, and remote program access to bit mapped screens.

The location of Mayflower services or of available Mayflower Operating Systems is done via Resource Manager. Remote module binding and configuration are done under program control, rather than at compile or link time, thus permitting a highly dynamic environment in which services may be fabricated on demand. This late binding allows the Mayflower system to go to Resource Manager to obtain the address of a remote service of the desired type, or to have one fabricated.

One experiment will be for some of the commands which the user types that may be CPU intensive (e.g. compilation or linking), not to be executed on his home machine but to result in a call to Resource Manager for a compiler, linker, etc. and a remote procedure call to be made to the resulting address to effect the action. For user applications which are not "installed" systems about which the Resource Manager knows, the home machine might request a Mayflower Operating System and specify to it the user's code which should be loaded and executed. Because Resource Manager does preloading, response to the user request could be almost instantaneous if the system as a whole was not too heavily loaded.

12.3. Visions of Sugarplums

In a distributed environment, the user may be executing several distinct tasks using many processes spread across various processors. A likely future scenario at Cambridge might involve a user working at a Dandelion workstation running the Xerox Development Environment (XDE).[1] Most editing would be done locally, as would tasks such as mail reading and calendar reminders. The user may access network resources to perform tasks not supported on the workstation or those sufficiently CPU intensive as to be more appropriately executed elsewhere. For example, the user may have a connection to Unix to access the wealth of applications it provides.

If the user was developing a Mayflower application, he might have connections to an interactive CLU compiler, a CLU linker/debugger, and a Mayflower operating system running his application. The CLU compiler would probably be running under Unix on a processor bank MicroVAX, and compiling code for 68000s. After editing the source on his workstation the user would issue a compilation command to the compiler. The source file would automatically be recalled by the fileserver when it was requested by the compiler, and the compiler would store the results of the compilation on the fileserver where they could be accessed by the linker or debugger. If the application was distributed, a remote debugger (which runs under Mayflower on the 68000s) might be used to debug the separate components.

The *User Secretary* is a command executive responsible for acquiring network resources for the user, and subsequently assisting the user in coordinating and controlling them. Preferably, the User Secretary would run in the workstation, though it will probably run on a remote Mayflower system because of the support Mayflower provides for distributed applications. The Secretary must determine what resources are needed, request them of the Resource Manager, and contact the resulting resources as appropriate. For example, a "print" command might result in access to a print server being requested from RM. The Secretary would pass the command arguments, which specify the names of the files to be printed along with formatting information, to the print server at the resulting address.

The interaction among tasks might be specified directly by the user if it was very simple (e.g. I/O redirection and piping in the Unix C shell). Alternatively, the coordination of resources might require task specific ***coordinator***

[1]XDE is a Mesa based programming environment for the Xerox Network System.

code in the Secretary. For example in bringing a CLU program object file up to date, the coordinator might begin with a consistency check based upon cluster procedure versions. If the distributed compilation system was used, one CLU compiler (with access to the CLU Library Server) might be acquired from RM on a non-preemptable basis, with several others on a preemptable basis. The coordinator would assign files to the different compilers, and then monitor their parallel execution. If there were compilation errors, certain of the compilations would have to be abandoned and the errors reported to the user. If some of the compilers were preempted, their compilations would have to be reassigned. Upon successful compilation, the coordinator would request and activate a CLU linker to produce the final program object file.

Files would take permanent residence on fileservers, with local store on workstations and processor bank machines being used for caching. Files would be stored and retrieved in entirety. If files were stored as structured, attributed objects (viz. Tioga [Cedar]), remote systems might have to access them through a filter. Directory servers would provide a directory structure for the global filing system, with indirection to appropriate fileservers at leaf nodes. They would implement access control, locking, file versions, and filter specification.

Remote processes would have access to the bitmapped screen of the workstation through a language independent interface. A call to a remote resource might result in its opening one or more windows on the user's screen. For example, the interactive CLU compiler might initially open a command executive window, and later open a separate window for error messages from compilations. The workstation software would support facilities such as windows, mouse handling, menus, command line/form parsing, and display of help strings. Among other benefits, this should encourage the use of more sophisticated displays and interactions with the user.

13. Summary and Conclusions

Thesis

The thesis of this dissertation is that sufficient commonality exists in the requirements for the management of distributed system resources that they can effectively be collected into a single resource management system. This minimizes the requirement for resource management mechanisms to be provided individually in external components, enforces a consistent interface for offering and requesting resources, and centralizes information necessary to implement policy incorporating multiple types and instances of resources. The cost of such generality is loss of the ability to tailor highly the management of resources with the result that they are less efficiently utilized. It is claimed that such cost is far outweighed by the advantages.

This research is concerned with high level resources available from components of the distributed system to users or other clients, such as an operating system, an edit session from an edit server, a compiler service, or a raw machine. A resource may be built upon one or more subresources, with such layering extending for multiple levels. For example, a compiler may run on an operating system which in turn runs on a machine.

The Processor Bank Philosophy

Much of the need for resource management, including the need to construct resources from subresources dynamically, stems from the use at Cambridge of a *processor bank*: a heterogeneous collection of machines which may be allocated on demand to run user services or network servers. A processor bank approach allows all of the users access to a very large amount of computing power which may be shared among them. Users may be allocated multiple machines for distributed applications or for concurrent execution of distinct tasks. Because the processor bank contains different types of machines it may support a wide variety of operating systems, languages, and tools. This facilitates importing existing software and allows the programmer to choose the most appropriate language and system for implementing new applications.

Two Resource Manager servers were built. The first was designed to support the original Cambridge Model Distributed System in which machines from the processor bank were allocated as personal computers for the duration of a login session. Given a system name, it could select an appropriate free machine and memory image from fixed lists, instruct a loading service to load the image into the given machine, and respond to a request from the newly loaded service with the location of the user's terminal.

The second Resource Manager, which is the major implementation effort associated with this research, takes a more general view of resources as objects available among components of the distributed system. It matches client requirements for a resource to those resources which it has been given, has been told are available, or has been told how to construct, by other network clients or by system managers. It has been in experimental use by Project Mayflower, but is not yet fully installed in the Cambridge distributed system.

Primary Resource Management Issues

The functionality of resources can be described by ***attributes*** chosen from a global namespace. Each resource has a class attribute (such as *CLU Compiler* or *68000*) and optionally several qualifying attributes (such as *VAX obj code* for a compiler, or *floppy disc drive* for a machine). The ***description*** of a resource also includes: factors differentiating it from other instances with similar functionality, data for accessing the resource, information about its structure, properties governing its allocation, and information about its current allocation.

Resource Manager maintains two data bases of information about resources available on the network. The ***Resource Repository*** contains information about all of the existing public resources. Repository entries are created either by clients placing resources in the public pool, or by Resource Manager after obtaining resources from servers or fabricators. The *Action Catalogue* contains ***resource actions*** which indicate how Resource Manager may obtain resources from servers or have them constructed from existing resources by fabricators. Each resource action describes the relation between two levels of a resource; obtaining a multiple level resource may involve executing several actions. Catalogue entries are created either by system managers, or by servers or fabricators willing to provide resources to the community on demand.

In satisfying client requests for resources, Resource Manager must determine combinations of existing resources and resource actions which will provide suitable resources. To do this it builds one or more ***blueprints*** or plans for construction. A blueprint may point directly to a reserved resource in the Resource Repository, or to a resource action and a list of blueprints for the required subresources. Resource Manager must then decide among these alternatives according to some resource management policy. Blueprints allow Resource Manager to be fairly certain that a particular plan for construction will succeed, and to ensure that all attribute requirements are met, before actually beginning construction.

There is a small set of general ***policies*** which are applied as appropriate to different types of resources. These are triggered by resource allocation properties (to be allocated, reusable, replenishable, support multiple) which are recorded in the Repository entries. Resource factors (quality, allocation time, cost, communication delay) are used to differentiate instances of a particular type of resource. The policy currently implemented utilizes a subset of these, with emphasis on minimum allocation time. The area of policy is probably the one I will pursue next, particularly the use of resource factors, preemption (the mechanisms for which are the same as for reclamation), and load balancing.

Additional Topics

The ***monitoring and reclamation*** of resources is handled by two mechanisms: one monitors aliveness by accepting refreshes of short timeouts from resources, and the other notifies interested parties when such timeouts expire so that they may act appropriately. Resource Manager requests to be notified of resource expiry so that it may delete the resource from its tables and adjust entries of related resources accordingly.

Authentication and *protection* mechanisms of the Cambridge distributed system are used to restrict the allocation of resources, as well as the entry and removal of resources or resource actions. Privilege lists within Resource Manager match attributes of resources or actions to privileges required of clients.

As part of Resource Manager's goal to respond to requests promptly, it keeps a stock of free resources ready for immediate allocation by obtaining resources from servers or fabricators ahead of time. It analyzes recent client requests for resources to determine which ones are likely to be the most popular.

When bootstrapping the distributed system, services must be brought up in an order consistent with their critical interdependence and services must be instantiated authentically. Special mechanisms are used to bootstrap the system through the level of Resource Manager, beyond which services can be instantiated by RM on demand.

In Conclusion

Resource Manager goes a considerable way in attaining the advantages of the processor bank philosophy. It provides a means of conversing about a wide variety of resources in a consistent manner. It matches requests for network resources to those which either already exist or can be constructed from existing ones. It detects and notifies appropriate parties of resource crashes. It provides authentication and protection both for clients requesting resources and for clients offering resources. And it provides resource information in a consistent form for implementing policy.

There is scope for further exploration, particularly of resource management policies. I anticipate that Resource Manager will continue to evolve as more experience with it is gained and as more demanding use of the processor bank is made.

References

[Accent 81]

R. F. Rashid and G. G. Robertson

Accent: A communication oriented network operating system kernel,

Carnegie-Mellon Univ.,

Proc. of the Eighth Symposium on Operating Systems Principles, Dec 1981, pp. 64-75.

[Argus 82]

B. Liskov and R. Scheifler,

Guardians and Actions: Linguistic Support for Robust, Distributed Programs,

M.I.T.,

Conf. Record of the Ninth Annual ACM Symposium on Principles of Programming Languages, 1982, pp. 7-19.

[Argus 84]

B. Liskov,

Overview of the Argus Language and System,

M.I.T.,

Programming Methodology Group Memo 40, Feb 1984.

[Birrell 80]

A. D. Birrell and R. M. Needham,

A Universal File Server,

Univ. of Cambridge,

IEEE Trans. on Software Engineering, Sep 1980, pp. 450-453.

[Boggs 82]

D. R. Boggs,

Internet Broadcasting,

Stanford Univ.,

PhD dissertation, Jan 1982, reprinted as Xerox Blue and White CSL-83-3.

Note: Reference labels were chosen for the information they could provide the reader at their occurance in the text.

[CAP 79]
M. V. Wilkes and R. M. Needham,
The Cambridge CAP Computer and Its Operating System,
Univ. of Cambridge,
North Holland, New York, 1979.

[Cedar]
The Cedar Reference Manual,
Xerox PARC,
internal document.

[Cedar 84]
W. Teitelman,
The Cedar Programming Environment: A Midterm Report and Examination,
Xerox PARC,
Blue and White CSL-83-11, Jun 1984.

[CLU 81]
B. Liskov et al.,
CLU Reference Manual,
M.I.T.,
Springer-Verlag, Lecture Notes in Computer Science #114, Heidelberg, 1981.

[CLU RPC 84]
K. G. Hamilton,
A Remote Procedure Call System,
Univ. of Cambridge,
PhD dissertation, Dec 1984.

[CMDS 80]
M. V. Wilkes, and R. M. Needham,
The Cambridge Model Distributed System,
Univ. of Cambridge,
Operating Systems Review, Jan 1980, pp. 21-29.

[Craft 83]
D. H. Craft,
Resource Management in a Decentralized System,
Univ. of Cambridge,
Proc. of the Ninth ACM Symposium on Operating Systems Principles, Oct 1983, pp. 11-19.

[Dannenberg 82]
R. B. Dannenberg,
Resource Sharing In A Network Of Personal Computers,
Carnegie-Mellon Univ.,
PhD dissertation, 1982.

[Dion 81]
J. Dion,
Reliable Storage in a Local Network,
Univ. of Cambridge,
PhD dissertation, Feb 1981, reprinted as Tech. Report #16.

[Distr Sys 81]
B. W. Lampson, M. Paul, and H. J. Siegert (eds.),
Distributed Systems - Architecture and Implementation, An Advanced Course,
Springer-Verlag, Lecture Notes in Computer Science #105, Heidelberg, 1981.

[Eden 83]
Guy T. Almes et al.,
The Eden System: A Technical Review,
Univ. of Washington,
Technical Report 83-10-05, Oct 1983.

[Garcia-Molina 82]
H. Garcia-Molina,
Elections in a Distributed Computing System,
Princeton Univ.,
IEEE Trans. on Computers, Jan 1982, pp. 48-59.

[Girling 82]
C. G. Girling,
Object Representation on a Heterogeneous Network,
Univ. of Cambridge,
Operating Systems Review, Oct 1982, pp. 49-59.

[Girling 83]
C. G. Girling,
Representation and Authentication on Computer Networks,
Univ. of Cambridge,
PhD dissertation, 1983.

[Jade 83]
I. H. Witten et al.,
JADE: A Distributed Software Prototyping Environment,
Univ. of Calgary,
Operating Systems Review, Jul 1983, pp. 10-13.

[Knight 82]
B. J. Knight,
Portable System Software for Personal Computers on a Network,
Univ. of Cambridge,
PhD dissertation, Apr 1982.

[Lampson 80]
B. W. Lampson and D. D. Redell,
Experience with Processes and Monitors in Mesa,
Xerox PARC and Xerox SDD,
Comm. ACM, Feb 1980, pp. 105-117.

[Lantz 80]
K. A. Lantz,
Uniform Interfaces for Distributed Systems: A Summary,
Univ. of Rochester,
Preprints for Workshop on Fundamental Issues of Distributed Computing, 1980, pp. 115-122.

[Lantz 82]
K. A. Lantz et al.,
Rochester's Intelligent Gateway,
Univ. of Rochester,
Computer, Oct 1982, pp. 54-68.

[Locus 83]
B. Walker et al.,
The LOCUS Distributed Operating System,
U.C.L.A.,
Proc. of the Ninth ACM Symposium on Operating Systems Principles, Oct 1983, pp. 49-70.

[Mayflower 84]
A. J. Herbert et al.,
The Mayflower Manifesto,
Univ. of Cambridge,
Systems Research Group, Project Mayflower note, Jun 1984.

[Medusa 80]
J. K. Ousterhout, D. A. Scelze, and P. S. Sindhu,
Medusa: An Experiment in Distributed Operating System Structure,
Carnegie-Mellon Univ.,
Comm. ACM, Feb 1980, pp. 92-105.

[Needham 82]
R. M. Needham and A. J. Herbert,
The Cambridge Distributed Computing System,
Univ. of Cambridge,
Addison-Wesley, London, 1982.

[NSW 77]
D. P. Geller,
The National Software Works - Access to Distributed Files and Tools,
Bolt, Bernek and Newman, Massachusetts Computer Associates, M.I.T., S.R.I. Intl., and U.C.L.A.,
Proc. of the 1977 ACM Annual Conference, pp. 39-43.

[Ody 84]
N. J. Ody,
Terminal Handling in a Distributed System,
Univ. of Cambridge,
PhD dissertation, Aug 1984.

[Popek 81]
G. Popek et al.,
LOCUS: A Network Transparent, High Reliability Distributed System,
U.C.L.A.,
Proc. of the Eighth Symposium on Operating Systems Principles, Dec 1981, pp. 169-177.

[Richards 79]
M. Richards et al.,
TRIPOS - A Portable Operating System for Minicomputers,
Univ. of Cambridge,
Software - Practice and Experience, Jul 1979, pp. 513-526.

[Richardson 84]
M. F. Richardson,
Filing System Services for Distributed Computer Systems,
Univ. of Cambridge,
PhD dissertation, Feb 1984.

[RSEXEC 73]
R. H. Thomas,
A resource sharing executive for the ARPANET,
Bolt, Bernek and Newman,
AFIPS Conf. Proc., 1973, pp. 155-163.

[StarOS 79]
A. K. Jones et al.,
StarOS, a Multiprocessor Operating System for the Support of Task Forces,
Carnegie-Mellon Univ.,
Proc. of the Seventh Symposium on Operating Systems Principles, Dec 1979, pp. 117-127.

[TriposFM 83]
M. F. Richardson and R. M. Needham,
The TRIPOS Filing Machine, a front end to a File Server,
Univ. of Cambridge,
Proc. of the Ninth ACM Symposium on Operating Systems Principles, Oct 1983, pp. 120-128.

[Wilkes 79]
M. V. Wilkes and D. J. Wheeler,
The Cambridge Digital Communication Ring,
Univ. of Cambridge,
Local Area Communications Network Symposium, May 1979.

[Worms 82]
J. F. Shoch and J. A. Hupp,
The 'Worm' Programs - Early Experience With A Distributed Computation,
Xerox PARC,
Comm. ACM, Mar 1982, pp. 172-180.

[Wulf 81]
W. A. Wulf, R. Levin, and S. P. Harbison,
HYDRA/C.mmp: An Experimental Computer System,
Carnegie-Mellon Univ.,
McGraw-Hill, New York, 1981.

[XDE 84]
Xerox Development Environment: Concepts & Principles
and **User's Guide**,
Xerox Office Systems Division, Palo Alto,
XDE3.0-1001 & 2001, Nov 1984.

[Yudkin 83]
M. Yudkin,
Resource Management in a Distributed System,
Univ. of Cambridge,
PhD dissertation, Sep 1983.

[Zahorjan 85]
J. Zahorjan,
Univ. of Washington,
personal communication concerning Eden, Jan 1985.