Distributed Parallel Processing

and the Factoring Problem

By

Brian Cox, B.Sc.

A dissertation presented in fulfilment of the requirements

for a M.Sc. Degree.

Supervisor

Dr. Michael Scott

School of Computer Applications

Dublin City University

September 1995

Declaration

I hereby certify that this material, which I now submit for assessment on the programme of study leading to the award of M.Sc. degree in Computer Science is entirely my own work and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the text of my work.

Signed: Brian Cox ID No.: 92700365

Brian Cox

Date: 28/9/'95

Acknowledgements

I would like to express my deep thanks to Dr. Michael Scott whose help, encouragement and supervision were invaluable.

Special thanks must go to my parents Ann and Tony, sister Ann-Marie, and girlfriend Bernadette for their encouragement, support and patience which helped me enormously.

I would also like to thank the School of Computer Applications for the opportunity of doing a M.Sc. and for their financial support.

Finally, I would like to thank all my friends in DCU for their help and a very enjoyable time - especially Brien, Colmán, Mike, and Ray, and the System Health Check group in Digital, Galway.

Distributed Parallel Processing and the Factoring Problem

Brian Cox B.Sc.

Abstract

This research is concerned with distributed parallel processing and how a computer cluster/network may be used to solve large and computationally expensive problems, specifically in the area of cryptography and the problem of factoring very large numbers.

Until recently few methods or systems were capable of harnessing the full potential power of a distributed environment. In order to realise the full potential of computer clusters, specially designed distributed parallel processing systems are needed.

Cryptography is the science of secure communications and has recently become commercially important and widely used. This research focuses on public key cryptography, the security of which is based on the difficulty of factoring extremely large numbers.

The research described in this thesis covers parallelism and distributed computing and describes an implementation of a distributed processing system. An introduction to cryptography is presented, followed by a discussion on factoring which centres on describing and implementing a distributed parallel version of Lenstra's Elliptic Curve factoring method.

Table Of Contents

Table of Figures

1. Introduction

Computers are not powerful enough, and never will be. No matter how fast and cheap they become, someone will always want more MIPS or FLOPS. General purpose machines draw power from their cheapness and flexibility - but not all general purpose computers are suited to the tasks they are given. Some problems are so demanding that they are effectively beyond ordinary computers - the sheer number of calculations required is so vast that they take too long to solve.

An example is the many-body problem in astrophysics. Calculating the interaction of the gravitational attraction between numerous masses demands huge computational resources. Realistic simulation of the dynamics of an astrophysical system should use a teraflops year of computing power; equivalent to a DEC Vax 11/780 running for a million years! To solve such problems in reasonable times, special purpose machines have been developed. However, very few have access to such machines, or the money to build or buy one. This thesis looks at a combination of old and new strategies for getting more MIPS from available chips.

1.1 The Hidden Cray Twin

Like most people I had seen a network as a way of passing information from one independent system to another - information such as electronic mail or data interchange between a client system and a database server system. However a network is a single system in its own right; a parallel processor. Hiding in every PC network there could be a computer as powerful as a CRAY [Bea90].

Supercomputers of the current generation are nearly all multi-processor systems. Some, like the latest CRAYs, use a small number of extremely powerful and fast processors working in parallel. Others, like the CONNECTION machine, use tens of thousands of simple and relatively slow microprocessors. All these supercomputers have in common the fact that their power derives from splitting a task between a number of processors. The number may vary but in every system processors are connected in such a way that data can be quickly passed between them. We can thus loosely define a parallel processor, and therefore a supercomputer, as consisting of a number of processors connected by an efficient high speed network.

This brings us back to our network of multi-tasking PCs. It's basic architecture is much the same as that of a parallel processor. The main difference is that network processors are physically far apart, so that communication between them is slower.

However, you can considerably increase the speed by carefully designing the pattern of network connections. A standard network is thus a basis for creating a parallel processing system many times more powerful than any single processor on the network.

Of course the parallel processor works only because the operating system is capable of dividing a task optimally between processors, and synchronising the operations involved. Unfortunately no parallel processing language and operating system for networks is commercially available, although a few experimental systems are being used in universities and research establishments. However, all the necessary communication and multi-tasking functions can be performed by operating systems such as Unix, OS/2 and Windows NT.

1.2 Overview of the Thesis

1.2.1 Aims and Objectives

The subject of this project - distributed parallel processing, is a specialised and complex area usually requiring special hardware and software. However, the most common, generally available parallel processing capable system is a local area network. With this in mind I set about designing and implementing a system, called DISTPROC, whereby programs could exploit the full power of a local area network.

1.2.2 Organisation of the Thesis

The first chapter introduces parallelism, the basic ideas, the problems involved, approaches taken and the possible benefits and costs involved.

Chapter two looks at distributed computing systems. The general design issues involved in developing a distributed system are covered and some distributed computing systems, such as PVM, are discussed.

This is followed by a description of the DISTPROC system. The objectives, design decisions, system architecture and interface are described in detail.

Chapter five covers cryptography, and in particular public-key cryptography, and its uses. Chapter six looks at factoring large numbers, the problems involved, and the algorithms used.

Chapter seven looks at how the DISTPROC system fared in factoring very large numbers using a distributed version of Lenstra's Elliptic Curve factoring algorithm.

1.3 References

[Bea90] Nick Beard PCW, Sept. 1990, Frontiers, page 247/8.

2. Parallelism

Computer performance has improved so fast that writers struggle to find clichés to apply. Supercomputer, however, is a relative term [Bea90]. It is reserved for the fastest machines of the day - today's supercomputer is tomorrow's sluggard. Today, supercomputing means parallel processing.

What sort of problems require today's supercomputers? A major application is simulation. Supercomputers find a place in nuclear weapon simulation, oceanography, weather prediction, geology, seismic activity analysis and special effects development in the film industry.

Some argue that sequential computers are pushing at the limits of physics and that future architectures must embrace parallelism, however not everyone agrees. Amdahl, a computer architect who ranks alongside Cray, says that 'demonstration is made of the continued validity of the single processor approach'. A crucial point is suitability. Not all styles of parallelism and computer are suited to all problems. The value of parallelism has been established for certain problem domains but not for others.

Any development in computing technology has to operate in a framework of existing systems. There are techniques, such as Amdahl's Law [Amd67], for measuring the speedup likely to be gained by any particular enhancement. Amdahl's Law states that the performance improvement gained by enhancing some portion of a system is limited by the fraction of execution time for which the enhancement is used. This is basically the law of diminishing returns; the cost of faster enhancements to the system is not always reflected in the overall benefit. Nevertheless, performance benefits are often achieved by gradually chipping away at whatever obstacle comes into view.

2.1 Types of Parallelism

There are many approaches to parallelism. The principal architectures are classified according to their instruction and data streams. Flynn's [Fly66] classification is the most common but is slightly dated at this stage.

2.1.1 SISD Architectures

A conventional computer is an SISD machine - single instruction stream, single data stream [Figure 1]. In conventional sequential programming languages a statement is executed and then control passes to the next statement in sequence. Various constructs exist to alter the sequence of execution, e.g. if, loops, functions, coroutines, etc., which means that most sequential programming languages are deterministic; that is, the order in which statements are executed is uniquely determined by the state of the program variables.

2.1.2 SIMD Architectures

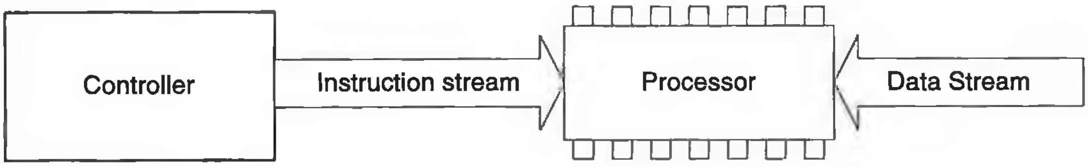


Figure 1. Model of a SISD computer

If a number of separate concurrent processors are operating on different data items, but doing the same thing to all of them, the computer is called a SIMD machine - single instruction stream, multiple data streams [Figure 2]; this covers processor arrays and pipeline processors.

2.1.2.1 Array Processing

Processor arrays consist of a number of identical processing units all under the control of a common control unit and correspond to Flynn's SIMD model. Each processing unit has

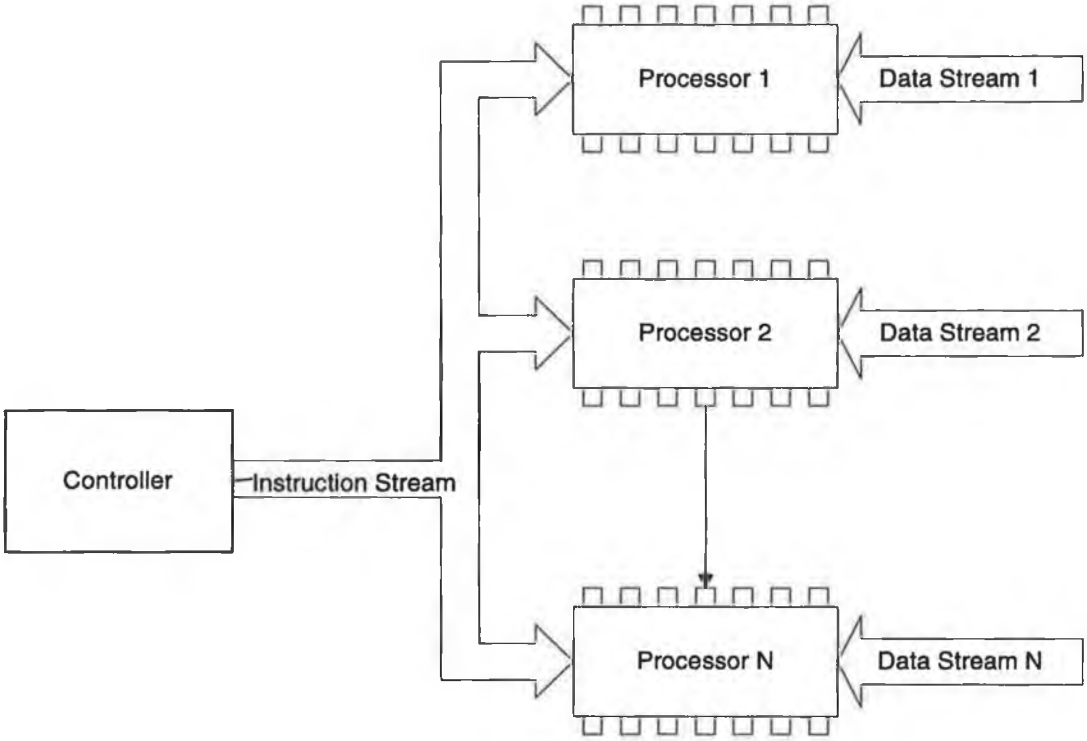


Figure 2. Model of a SIMD computer

memory associated with it and access to its own data items; therefore the same operation can be performed on many data items simultaneously. The speed of transfer and communication between processors, host and memory will influence the performance capabilities of an architecture [Hoc88].

2.1.2.2 Pipeline or Vector Processing

In a pipeline system, arithmetical operations are split into successive stages, and separate chunks of hardware used to attack each stage. These machines are also known as vector processors, because they are efficient only when arithmetic operations to be performed are vectorised - turned into continuous streams of data. A vector is a group of numbers of a similar type that can be treated simultaneously by the machine, as opposed to scalar values which are single quantities. Scalar operations affect pairs of data items, and are better suited to conventional architectures. Vector processors can be remarkably fast when applicable, but unfortunately many important applications cannot be vectorised. The overhead of using a pipeline is the pipe priming time - the time it takes for the pipe to fill, so the more consecutive times a pipeline is used, the better the performance.

2.1.3 MIMD Architectures

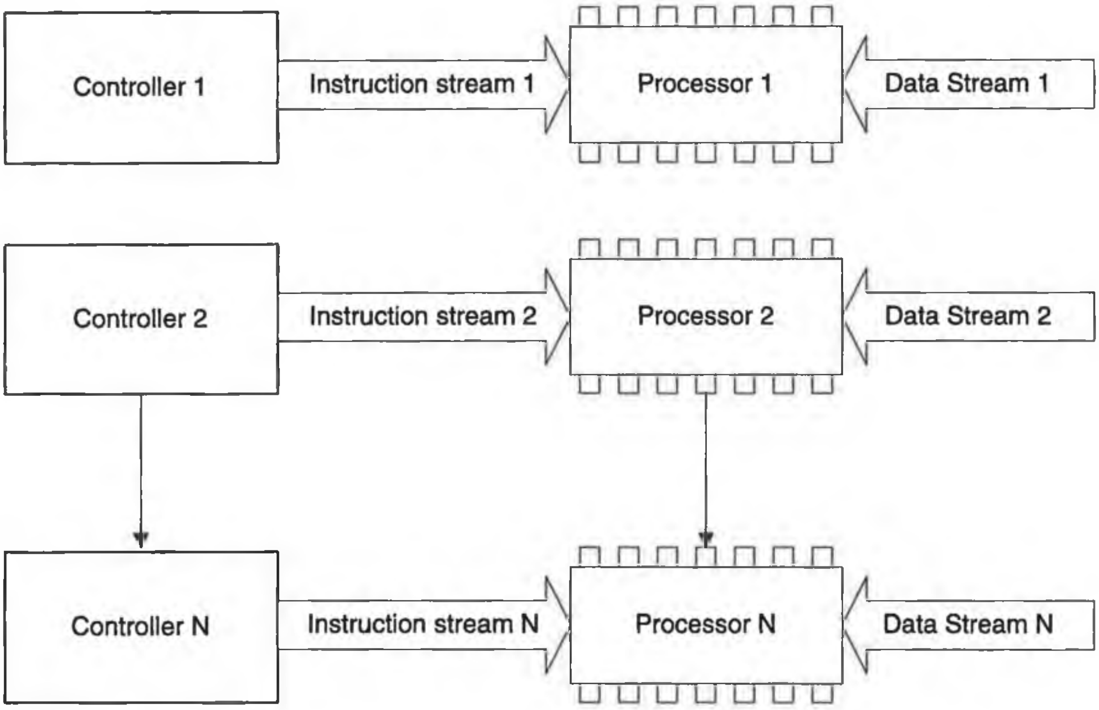


Figure 3. Model of a MIMD computer

A third approach is the more complex MIMD - multiple instruction streams, multiple data streams - machine [Figure 3]. Here each processing element has a control unit in addition to an arithmetical logic unit and memory. Each element of the network can function as a fully fledged digital computer, acting in concert to amplify the power of the group. The MIMD model is used to describe both tightly and loosely coupled processors working on a single problem. It does not allow for the interconnection topology or the mechanism of sharing and protecting information.

Another approach to parallel supercomputing is the multicomputer method, which really took off only in the mid '80s. This is based on networks of computers, and is much less expensive than 'conventional' supercomputing.

2.1.3.1 Shared Memory Processing

A shared memory machine consists of a number of processors all having access to a single shared memory [Figure 4]. This makes communication between processes easy in principle, however when there are many fast CPUs vying for access to the same memory, bus contention can bring the system to its knees. To alleviate this caches are often added to provide each processor with its own private memory for copies of live variables; unfortunately this introduces the problem of cache coherency. These problems limit the number of processors you can usefully put into a bus shared memory computer (i.e.,

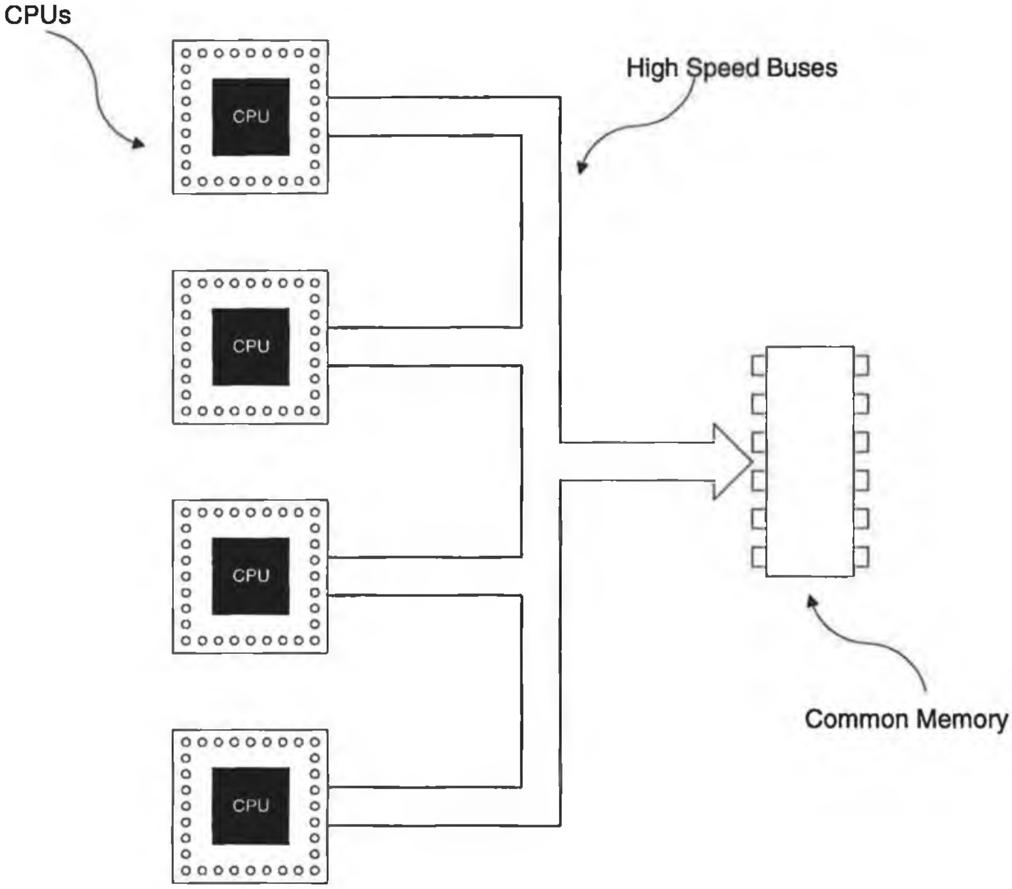


Figure 4. Shared Memory MIMD, Tightly Coupled System

scalability) to tens rather than hundreds.

Shared memory processes are usually controlled by the Fork / Join or Cobegin statements. Techniques for synchronisation must be used to protect the integrity of information shared between processes; mechanisms used include semaphores and monitors. Variants on the shared memory architecture are sometimes classified as PRAM (parallel random access machines).

2.1.3.2 Distributed Memory Processing

In a distributed memory MIMD machine, every computing node is a complete computer, with its own local memory [Figure 5]. These machines are often referred to as multicomputers. Since there is no shared common memory between the nodes, results must be passed between nodes over a communication network. Distributed memory MIMD architectures are also

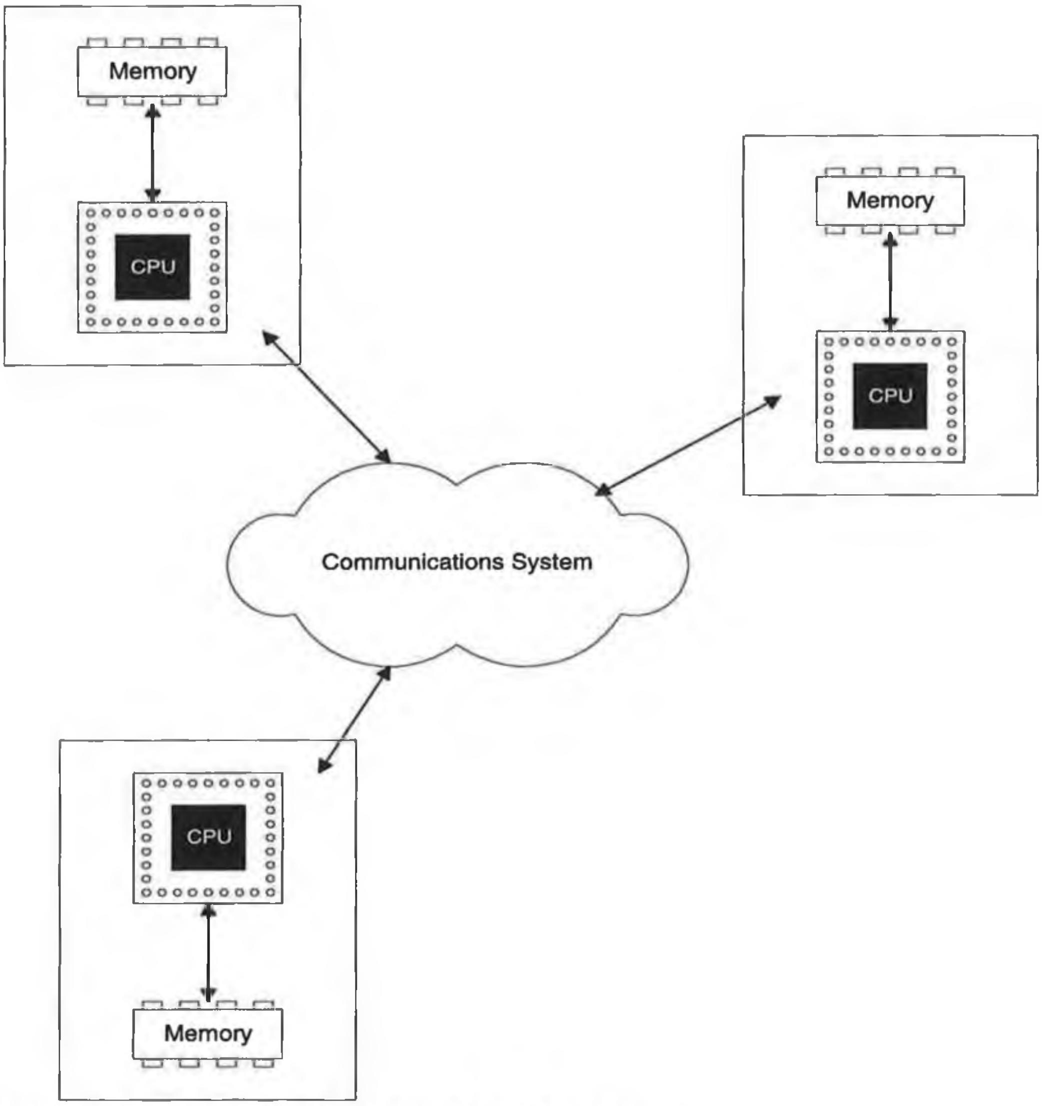


Figure 5. Distributed Memory MIMD, Loosely Coupled System

called message-passing architectures.

The performance of a multicomputer is as much affected by the speed of its communication network as it is by the speed of the processing elements. The balance between communication time and computation time in a message passing machine varies according to the problem, the algorithm used and the topology and performance of the communications network used. Applying more processors to a problem on a message passing computer does not ensure improved performance because the machine may spend all its time communicating while half its processors lie idle.

2.2 Synchronisation

In a shared memory environment great care must be taken when two or more processes simultaneously access a shared variable. Sections of code that need to be protected are known as critical sections, with controlled access enforced through mutual exclusion which treats sections of code as an indivisible operation. Semaphores and monitors are among a number of mechanisms proposed to achieve mutual exclusion in a shared memory architecture.

In a distributed memory environment synchronisation can be achieved implicitly through the use of synchronous message passing which will be discussed later.

2.2.1 Semaphores

Dijkstra [Dij65] originally proposed semaphores as a synchronisation mechanism. A semaphore's value can only be accessed and altered by the operations P and V and an initialisation operation. Binary semaphores can assume only the value 0 or 1. General semaphores (also known as counting semaphores) can assume only non-negative integer values.

The operations P and V are indivisible. Mutual exclusion on the semaphore, S, is enforced within P(S) and V(S). If several processes attempt a P(S) simultaneously, only one will be allowed to proceed. The others will be kept waiting, but the implementation of P and V guarantee that processes will not suffer indefinite postponement.

2.2.2 Monitors

A monitor, introduced by Dijkstra [Dij71], is a concurrency construct that consists of a set of permanent variables, a set of procedures and a body (i.e. a sequence of statements) and is used to control allocation access to a resource. The body is executed when the program is initiated and provides initial values for the monitor variables. Thereafter the monitor is only accessed via its procedures. The underlying system schedules the execution of the monitor's procedures. Access to these procedures is granted to only selected processes. Since mutual exclusion is guaranteed, concurrency problems, such as indeterminate outcomes, are avoided. When a monitor is used it guarantees that the initialisation code will be executed before any contention can occur, and only one procedure will be executed at any one time.

2.3 Message Passing

With the trend towards distributed systems, there has been a surge of interest in message-based interprocess communications, resulting in standards such as the Message Passing Interface [MPI].

In a message-passing programming environment processes share data by explicitly sending it from one process to another by means of channels. This can be viewed as extending semaphores to convey data as well as synchronisation, or as an extension of the shared memory processing model where processors only have private memory. Either synchronous (blocking) or asynchronous (non-blocking) message passing can be used.

There are a few disadvantages of message passing systems. Communications in a distributed system pose serious security problems, such as the *authentication problem* which deals with verifying the identity of the remote communications entity. There is also the possibility that transmissions can be flawed and even lost so an acknowledgement protocol must be used to ensure each transmission succeeds.

2.3.1 Communications Schemes

A number of communications schemes have been proposed including:

- direct naming
- global mailboxes
- channel naming

Naming each process unambiguously is an additional complication in distributed systems. Process creation and destruction can be co-ordinated through some centralised naming mechanism, but this introduces considerable transmission overhead. The most viable option is to have each computer in a distributed system use a unique name; now processes can be identified by a combination of the computer name and process name.

A *mailbox* is a message queue that may be used by multiple senders and receivers and acts as an intermediary. The advantage of global mailboxes is also the disadvantage; the sender does not have control over who will receive the message. In a distributed system, the receivers can be dispersed across many computers, thus requiring two transmissions for most of the messages. This problem is solved by giving each receiver a *port*; this is defined as a mailbox used by multiple senders and a single receiver. In distributed systems, each server ordinarily has a port at which it receives requests from many clients.

Channels are used to link two processes in one direction (e.g. send x via channel 2) and can be either static or dynamic.

2.3.2 Communications Patterns

Message passing implies the co-operation of the sender and receiver, but there are a number of possible patterns. The pattern in which processes are distributed throughout the system and the means used to identified them dictates which of these communications patterns can be used.

One to One: Both parties are specified, by using their names, or defining a distinct channel between them, or defining them as the only parties who can use this mailbox. When two processes pass messages at several points [Figure 6] matching on a particular communication cannot be achieved by process naming. Matching communications can be achieved by channel naming or mailboxes as these can be defined for separate transactions.

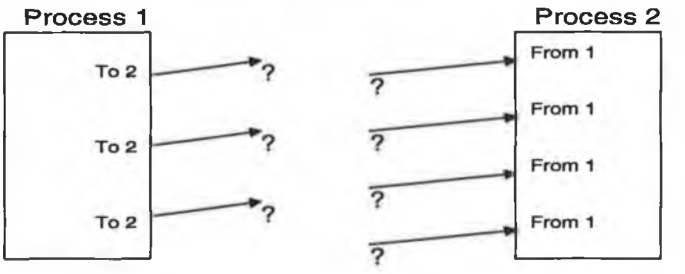


Figure 6: Matching communications?

Many to One: A single process is prepared to accept messages from any of a set of processes. Thus there are still only two parties to a transaction, but the sender can be one of many.

One to Many: In this case there is one sender and many potential receivers.

Many to Many: In this case there are many potential receivers and many potential senders with either one or all of the receivers accepting communications from one or any one of the senders or else some combination of receivers and senders will agree on a communication. Such a strategy is difficult to implement, and with multiple clients and multiple servers will closely resemble the shared memory processing model.

2.3.3 Synchronous Message Passing

Synchronous message passing makes both the sender and receiver wait until both sides are ready to proceed. Once the message has been exchanged both sides can continue. Advantages of synchronous message passing are that there is no need for buffering and the state of the corresponding process is more readily available.

Synchronised, zero-buffered communication greatly simplifies programming and can be efficiently implemented. Zero buffered communication eliminates the need for message buffers and queues. Synchronised communication prevents accidental loss of data arising from programming errors, e.g forgetting to wait for a reply. This form of communication means that one process must wait for the other. Synchronisation is achieved when one task is ready to execute an input primitive and the second process is ready to execute an output primitive.

2.3.4 Asynchronous Message Passing

In asynchronous message passing the sender initiates the message and then continues with its processing. The message is accepted by the underlying system and some time later is delivered to its destination. A process awaiting a message will be delayed until the message arrives. Such a framework was described by [Whi87] and implemented as the language PLITS (Programming Language In The Sky).

Messages may not be received in the order that they were sent, but this may be a benefit as important messages can be prioritised. Other disadvantages are the necessity of large buffers for storing unreceived messages, as well the lack of an immediate acknowledgement of message receipt. The receiver can explicitly acknowledge the receipt of a message, but then should the original sender be expected to acknowledge the acknowledgement?

2.4 Parallel Programming

For every parallel system there is a wide range of programming models and languages. We will now look at some popular parallel programming models.

2.4.1 Parallel Constructs

Several approaches have been introduced to allow users to declare explicitly which sections of a program can be executed in parallel on shared memory processors [Die90], with the two most popular being Fork / Join, and Cobegin.

Structured languages are better served with a structured parallel operator rather than a loosely structured Fork / Join. Structured languages are also more able to support the use of variables local to a process, which is useful in shared memory architectures that have additional cache or private memory available. Cobegins are also easily nested whereas Fork / Join are not as flexible in this regard.

2.4.1.1 Fork and Join

Anderson [And65] suggested two primitives called Fork, which starts a parallel process, and Join, which waits for parallel processes to finish (it synchronises processes). These calls correspond to the fork() and wait() system calls in the Unix operating systems.

A Fork call is similar to a procedure, allowing another process (routine) to start, while allowing the calling process to continue execution. The calling process must at some further stage contain a Join, where it will wait until the new process has terminated, which may already have happened due to the very nature of parallel processing.

2.4.1.2 Process Declaration

A variation on the Fork / Join model is based on process declarations. When processes are created they can be considered to be in a state of suspended animation. Such processes can be brought to life by a call such as Fork. In Concurrent Pascal this explicit animation can only be used during program initialisation, therefore giving a fixed number of processes. ADA uses TASKs (its version of processes) which allow a variable number of processes at run time.

2.4.1.3 Cobegin or the Parallel Clause

The Cobegin or parallel clause (par) provides a structured means of specifying parallel execution of a set of statements. The Cobegin terminates only when all the constituent tasks have completed. This single -entry -exit control structure allows specification of most concurrent computations while maintaining ease of readability.

```
par begin
        statement 1;
        statement 2;
          :
        statement n;
par end
```

All statements can be executed in any order or in parallel, and all must finish before the program can continue past the **end** statement.

2.4.2 Communicating Sequential Processes

The CSP approach to parallel programming, based on the original ideas developed by Hoare [Hoa85], assumes processes run sequentially. CSP has had a major impact on concurrent programming ideas and research because of its conceptual simplicity and potential for efficient implementation. The concepts of synchronisation and communication are unified in CSP. This model has been adopted by two major languages, Occam and Ada, that incorporate facilities for representing concurrent activities.

2.4.3 The Occam Model of Process Parallelism

Occam, developed by INMOS [INM88], is based around the ideas of CSP and allows advantage to be taken of the concurrent processing capabilities that are available in the INMOS transputer system.

One of the main aims of Occam was to remove the problems caused by using shared memory as a means to implement process communications, as such systems become less efficient as the number of communicating processes increase. It achieves this by allowing an application to be expressed in terms of a number of concurrent processes which communicate using channels, (localised processing and communication).

A process [Figure 7] may be defined as being carried out sequentially or in parallel with respect to other defined constructs. It may be used as part of a larger construct thereby allowing a hierarchical decomposition of applications to occur naturally. Each channel provides a one-way communication between two constructs. Communications between the constructs is synchronised, i.e. the first process which is ready to communicate waits until the second is also ready; when both are ready the exchange can take place.

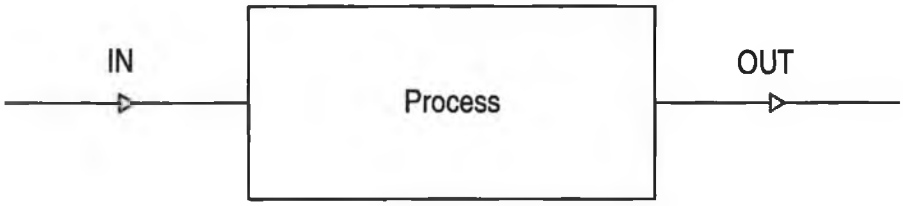


Figure 7. Description of a process in Occam. The process has an input channel called IN and an output channel called OUT

2.4.4 Adding Parallelism - Linda

Linda [Car89] consists of a small set of operations that can be added to any language to allow parallel processing. The concept of Linda is based on the tuple space model of parallel processing. Processes and data can be considered to be objects floating in tuple space. Sender and receiver processes communicate by the sender creating a data object that it releases into the tuple space; the receiver can examine such objects, hence the communication takes place. Processes are generated in a similar manner, only they are considered to be live tuples that can carry out their own process and then become data objects. Linda appears to be able to support most models of parallelism, excluding some pipeline processing and array models. There are four basic operations defined in Linda:

1. **in** - removes and reads a tuple from the tuple space.
2. **rd** - reads a tuple without removing it from the tuple space..
3. **out** - adds a tuple to the tuple space.
4. **eval** - evaluates an object.

Tuples exist independent of the creators and are matched by element type and position. Creation of tuples is non-blocking but reading will block if there is no match, and where there are a number of matches a non-deterministic choice will be made between the matching tuples. Tuples can be grouped together to yield a variety of data structures.

2.4.5 Object-Oriented Programming

The object-oriented model is based on the concept of encapsulating data and its operations together into an object [Boo91]. Therefore, the object-oriented approach is well suited to decentralised systems, as each instance of an object can be thought of as a separate process which can be allocated to a processor. Objects are activated when they receive a message, which relates to the message passing model. Some experimental systems include POOL (Parallel Object Oriented Language) [Ame87] and Mentat [Gri93].

2.4.6 Other Models

There are many alternative models to the basic models. The functional model concentrates on what the program is to do rather than how to do it and can be split into two parts; applicative programming models based on the operations of functions, and dataflow models where programs are driven by the need for data to flow through the system. Logic programming models are based on symbolic processing where parallelism is found in four ways: OR-parallelism, AND-parallelism, Stream parallelism or Unification parallelism.

2.5 Summary

For several years parallel processing was in danger of joining that not so exclusive club of can't miss technologies. The difficulty of parallel software development caused widespread disillusionment. This has turned around somewhat with the remarkable progress in processors and advances in the understanding of communications networks to produce a technology that can solve real world problems.

There are still many issues to be addressed, such as the diversity of parallel models and architectures. There is much debate as to whether a unified parallel architecture can be achieved or if it is totally desirable, but it should be followed up as a means of defining an ideal machine that is mapable onto all real parallel hardware. Other areas for improvement include support tools and compilers, which are currently not up to the task of automatic translations and must rely on guidance from the programmer.

2.6 References

[Amd67] Amdahl, G.M., “The Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities,” in AFIPS Conf. Proc., 1967.

[Ame87] America, P., “POOL,”, Object-Oriented Concurrent Programming, A. Yonezawa (Ed.), MIT Press, 1987.

[Bea90] Beard, N., “Frontiers,” Personal Computer World, Sept. 1990.

[Boo91] Booch, Grady, “Object Oriented Design with Applications,” Benjamin Cummins, 1991.

[Car89] Carriero, N., Gelernter, D., “Linda in Context,” C.A.C.M., Vol. 32, pp. 444-458, 1989.

[Dij65] Dijkstra, E. W., “Cooperating Sequential Processes,” Technological University, Eindhoven, Netherlands, 1965. (Reprinted in F. Genuys (ed.), *Programming Languages, Academic Press, New York, 1968, pp. 43-112).*

[Dij71] Dijkstra, E. W., “Hierarchical Ordering of Sequential Processes,” Acta Informatica, Vol. 1, 1971, pp. 115-138.

[Die90] Dietel, H.M., "Operating Systems", 2nd Ed., Addison-Wesley, 1990.

[Fly66] Flynn, M. J., “Very High Speed Computing Systems”, Proceedings of the IEEE, Vol. 54, pp. 1901-1909.

[Gri93] Grimshaw, A.S., “Easy to use Object Oriented Parallel Processing with Mentat,” IEEE Computer, May 1993.

[Hoa85] Hoare, C. A. R., ”Communicating Sequential Processes,” Prentice Hall, 1985.

[Hoc88] Hockney, R., and C. Jeshope, “Parallel Computers 2,” Adam Hilger, 1988.

[INM88] INMOS, “Occam 2 Reference Manual,” Prentice Hall, 1988.

[MPI93] Message Passing Interface Forum. Document for a Standard Message-Passing Interface. Technical Report No. CS-93-214., University of Tennessee, Nov. 1993. Send a mail message to **netlib@ornl.gov** containing the line **send index from MPI,** or use anonymous ftp from **netlib2.cs.utk.edu**.

[Whi87] Whiddet, D., "Concurrent Programming for Software Engineers," Ellis Horwood, 1987

3. Distributed Computing

3.1 Introduction

'*Distributed computing systems*' can mean almost anything in the language of computing. The meaning intended here is that it consists of multiple independent processors that co-operate by message passing over a communications network. So, we are concentrating on multicomputers, workstation LANs and workstation WANs. Distributed computing systems are gradually evolving and research on architectures and interconnection networks has resulted in low cost distributed systems with large numbers of powerful processors that can communicate at high speeds.

'*Distributed computing*' is coercing autonomous, networked computers to work simultaneously on a set of problems in a co-operative manner [Bal90]. Such systems are becoming commonplace and are now available to many potential users and used for many different types of application. Distributed systems are favoured over other architectures, such as uniprocessors or shared-memory multiprocessors, because they can provide better computer performance at a lower cost, utilise idle cycles and resources, increase reliability and availability, match applications to hardware (inherently distributed) or hardware to applications (special requirements).

We'll first look at some of the issues that must be addressed when dealing with distributed systems, followed by a look at various systems specifically targeted at supporting distributed computing. We'll finish with an introduction to the object oriented approach to distributed systems and computing.

Frequently, distributed computing systems are used for running different jobs or services on different machines, however this thesis is concerned with how multiple computers can co-operate in executing single jobs. In particular we look at how the turn-around time of a job can be reduced by running different parts of a job in parallel on multiple machines.

3.1.1 Distributed Computing Systems

Distributed computing systems can be classified according to their interconnection networks. Traditionally, a distributed architecture in which communication is fast and reliable and where processors are physically close to one another is said to be *closely / tightly coupled*; systems with slow and unreliable communication between processors that are physically dispersed are termed *loosely coupled*.

Closely coupled distributed systems use a communications network consisting of fast, reliable, point-to-point links, which connect each processor to some subset of the other processors (e.g. the hypercubes [Ran88], Transputer networks [May84]).

A more loosely coupled type of distributed system is a workstation LAN, where direct communication between any two processors is allowed. Communication in many LANs is not always reliable and a message may be damaged, arrive out of order, or not arrive at its destination at all; therefore software protocols must be used to implement reliable communications. LANs are limited to relatively short distances, so to interconnect processors that are further apart a wide area network (WAN) is needed; this can be viewed as a very loosely coupled distributed system. Communication in a WAN is slower and less reliable than in a LAN, but the distinctions will be reduced with the increased availability of newer high speed lines.

There is a wide range of distributed systems, ranging from closely coupled to very loosely coupled. Regardless, all systems fit into the same model, i.e. autonomous processors connected by some kind of message passing network..

3.1.2 Distributed Computing

Achieving speedup through parallelism is a common reason for running an application on a distributed computing system. By executing different parts of a program on different processors at the same time some programs will execute faster and finish sooner. These applications can be run just as well on shared-memory multiprocessors, but there is trouble scaling up to large numbers of processors, which explains the interest in implementing parallel programs on distributed systems.

Parallel applications can be classified by the grain of parallelism used, which is measured as the ratio of computation to communication. Large-grain parallel programs consist mostly of computation with little communication; fine-grain parallel programs communicate more frequently; medium grain parallel programs are in between. The grain size refers to the nature of the application, whereas coupling describes the architecture.

Fine-grain and medium-grain parallelism are most suited to closely coupled distributed systems, as the communications overhead in loosely coupled systems is usually prohibitively expensive. Large-grain parallelism is suitable for both closely and loosely coupled distributed systems. Numerous applications can benefit from large-grain parallelism, [Ran88][Ath88], such as heuristic search algorithms.

3.1.3 Design Issues in Distributed Computing

Within a distributed computing system many issues arise that are more difficult to address than in single-workstation computing, such as load balancing, scalability, security, fault tolerance, data distribution / locality, distributed I/O, programming environment, and debugging. Sometimes the local workstation operating system makes addressing these issues arbitrarily difficult, suggesting the need to make workstation operating systems aware of their distributed computing environment.

The principal requirements for a distributed computing environment are that applications must be able to locate the processing capability it requires, send and receive parameters and results which must be meaningful on various machine architectures within the environment, and a process must be useable across platforms and languages. Other requirements include security, interface description, and how information is transmitted. These are the absolute basics without which distributed computing will not work well.

3.1.4 Approaches to Distributed Computing

Programming support for implementing distributed applications may be provided by a variety of levels ranging from the (distributed) operating system to a language especially designed for distributed programming.

We will ignore the approach where the application is built directly on top of the hardware, which although it provides total control over all primitives provided by the hardware, is highly hardware-dependent and labour intensive.

The first approach that is usually considered is based on the operating system which can be a nucleus (providing only processes and interprocess communication), a network operating system, or a full-blown distributed operating system. With this approach, sequential languages are used along with a collection of operating system primitives accessed through library routines. Disadvantages of this approach are that the control structures and data types of sequential languages are often inadequate for distributed programming.

Another approach uses a programming language with all the necessary constructs for distributed programs. This shields the programmer from both the hardware and operating system and can also present a higher level model of a distributed system. Language level support for distributed programming can overcome these disadvantages while providing improved readability, portability, and static type checking. A language may also present a programming model that is higher level and more abstract than the message-passing model supported by most operating systems.

The approach that we will concentrate on consists of systems specifically designed to enable distributed parallel processing on top of a network of autonomous workstations. With such an approach we can program to a higher level than that provided by a distributed operating system, in a more consistent and complete environment created with parallelism in mind. However, this method does not overcome the disadvantages associated with extending or augmenting sequential languages.

We will also consider the latest approach to distributed computing, distributed/interoperable objects. Objects can hide much of the complexity involved in developing distributed applications, leaving the programmer to concentrate on the core design and implementation rather than ancillary details.

3.1.5 Programming Issues in Distributed Computing

There are basically three issues that distinguish distributed programming from sequential programming:

1. The use of multiple processors.
2. The co-operation among the processors.
3. The potential for partial failure.

Distributed programs execute pieces of their code in parallel on different processors. In high-performance applications, where the goal is to make the best possible use of the available processors to achieve maximum speedup, the decision as to which computations to run in parallel is of great importance. In fault-tolerant applications, decisions to perform functions on different processors are based on increasing reliability or availability. For special-function and inherently distributed applications, functions may be performed on a given processor because it has certain capabilities or contains needed data. The first requirement for distributed programming support is the ability to assign different parts of a program to different processors.

The processors of a distributed system need to co-operate while executing a distributed application. With parallel applications, processors sometimes have to exchange intermediate results, and synchronise their actions. Therefore the second requirement for programming distributed systems is that processes are able to communicate and synchronise.

In a distributed system some processors may fail while others continue. This property can be used to write programs that can tolerate hardware failures and is particularly important for fault-tolerant applications. Therefore, the third requirement for distributed programming support, is the ability to detect and recover from partial failure of the system.

3.2 Design Issues

In this section we will look at some of the various design issues that arise when designing and implementing a distributed computing system. Many of the areas covered are relevant across the spectrum of distributed systems.

3.2.1 Communication and Synchronisation

Primitives that allow processes to exchange information and synchronise their actions should be provided by all distributed systems. The chosen mechanism is usually based on the message passing model, although other models such as the RPC (Remote Procedure Call) [Tan88] and transactions are also popular. Once the model is chosen several additional issues and decisions must be considered for a complete communication and synchronisation service.

Reliable communication is not usually provided by the underlying hardware, and must be provided by the system. Message passing can be made reliable by adding software protocols, which are included in the system kernel by most distributed operating systems.

Synchronous (blocking) and asynchronous (non-blocking) messaging passing must also be considered. Synchronous message passing systems, such as RPC, are easier to use, but might reduce the possible level of parallelism.

Messages are usually considered as a sequence of bytes by the operating system with no regard to the contents. This means that the sender and receiver of a message must agree on the form and content of the message.

A certain degree of non-determinism should be supported by the communications primitives, whereby a receiver can receive a message from any of a set of processes, with support for multicast and forwarding of messages being desirable.

3.2.2 Process Management

Process management deals with the distributed systems mechanisms and policies for sharing processing resources spread around a network among all processes, and in particular to perform local and remote operations on processes. In an ideal case all operations on remote processes should be transparent to the user.

The following operations should be supported remotely: create, destroy, suspend, resume, and run processes. The major problem in remote execution of a process is the selection of an execution site, but this is a policy decision and should be decided by the resource management system.

Another important part of process management is process migration, which is crucial to distributed scheduling (load balancing). A process migration policy is needed to define when to migrate which process where, but once again this belongs in the resource management system.

There are four basic models for process interaction in a distributed computing environment, namely:

1. The client-server model.
2. The integrated model.
3. The pipe mode - based on the concept of a process.
4. A remote procedure call - which allows a process to call a procedure at a remote computer.

The majority of distributed systems fall into the client-server model. In this model control is distributed among the various processes in the system, with many of the system's functions implemented in user processes. Processes are generally classified as being either clients or servers. To request a service, a user (client) process sends a request to a server process, which then does the work and returns a response. In this model, control of individual resources is centralised in a server. There are three major problems with the client-server model:

Control of individual resources is centralised in a single server. This means that if a server fails then that element of control fails, which is unacceptable if that server is critical to the operation of the system.

1. Each server is a potential bottleneck, which worsens as more clients are added.
2. To improve performance multiple implementations of similar functions can be used, however this adds to the cost of a distributed system. Local caching can help overcome this problem somewhat.

Solutions to these problems have been proposed, such as multiple resource managers. Many of these solutions just move these problems to other areas. These problems led to the integrated model, where each computer's software is designed as a complete facility with a general file system and name interpretation mechanisms, on which LOCUS [Wal83] was based.

3.2.2.1 Data Marshalling / Persistence (Serialisation)

If a data structure is passed as a value parameter in a fork or operation invocation it has to be placed (marshalled) into a network packet, which is usually a linear sequence of bytes. Also, any structured values returned by a remote operation - either as result or output parameter, have to be unmarshalled. Finally, if the run time system decides to create a copy of a shared data object on a certain processor, the current value of the object has to be sent, which may also require marshalling.

For data structures that are represented as contiguous memory blocks, the above tasks are simple. Unfortunately, data structures whose size can change dynamically (e.g., graphs, arrays, sets) are hard to implement efficiently using a single contiguous memory block. In general data structures consist of multiple blocks of memory, linked through pointers. The run time system therefore needs to have information about the representation of structured variables (i.e., variables of a structured type), in order to carry out each of the above tasks.

3.2.3 Resource Allocation

Resource management is one of the basic and most important problems of distributed systems. Resource management in distributed systems consists of three issues, namely resource allocation, deadlock detection and resolution, and resource protection. In this section we will look at resource allocation.

Resource allocation can be managed by server based resource managers or agent based resource managers. Management systems allocating resources to processes can be divided into two groups. The first group consists of those systems which provide a variety of resources. The second group contains those systems which provide access to a single type of resource, and is mainly oriented towards processor / workstation allocation.

One of the basic resources managed in any computer system, particularly in a distributed system, are processors so we will now look at distributed scheduling.

3.2.3.1 Load Distribution - Sharing and Balancing

A distributed scheduling policy consists of two components:

1. A local scheduling discipline which determines the allocation of a processor among its local processes.

2. A load distributing strategy that is responsible for the distribution of the system workload among the computers of a distributed system by means of process migration and remote execution.

Since load scheduling in a distributed system is no different to that in a normal operating system we will ignore it here and concentrate instead on load distributing strategies. Approaches to the problem of load distribution can be divided in two groups, load sharing and load balancing.

Allowing processes to get computation service on idle workstations is called load sharing. Load sharing, also known as processor allocation or hunting for an idle processor, can be performed on the basis of many different organisations imposed on a set of available processors. Processors can be organised in a logical hierarchy, in a logical ring, or no special structure. Load sharing provides a local solution, as it improves performance indices of only a few users and their processes.

Load balancing algorithms on the other hand, strive to equalise the system workload among all computers of a distributed system and improve performance indices globally. Load balancing (distributed scheduling) algorithms can be grouped into task placement algorithms and dynamic load balancing. The former algorithm finds the optimal location for all processes before they start execution (static load balancing), while the latter includes algorithms which allow processes to migrate to remote computers once they have begun execution and is associated with process migration.

Scheduling processes on processors is an important issue in the design of a distributed system. Schedulers should try to spread the load of the system evenly over the available processors and also give each process a fair share of the processor cycles. Strategies for scheduling should consider both local and global scheduling. Global scheduling determines which processes are to be run on which processors while local scheduling of a single processor determines which of the competing processes is to be run at a given point in time.

3.2.4 Protection and Security

One of the main aspects of distributed systems is the sharing of resources between certain users. However, not all users are allowed access to all resources. Moreover, there could be users trying to eavesdrop or tamper with messages, or attempting to masquerade as someone else. This demonstrates the need for security in a distributed system. Distributed system security is a vast topic which covers not only such advanced topics as access control, communication security and authentication measures, but also more conventional measures such as physical, procedural and personal security.

To deal with all aspects of security a distributed system must provide resource protection (authorisation), secure communication, and user authentication. Resource protection involves protecting objects in the distributed system from unauthorised access. Communication security safeguards information transmitted between computers connected by communication media. Authentication assures the receiver of a message that it came from the reputed source and that it has not been changed since.

Methods for achieving these goals are based on the use of cryptography to protect messages and communications along with an authentication service to enable clients, servers and other communication partners to verify each others identities.

3.2.4.1 Kerberos

One of the best known security services is Kerberos which is an encryption-based system designed to authenticate users and network connections. It was developed at MIT's Project Athena in the 1980's and is used to prevent unauthorised access. It does this so well that it is now almost a de facto standard for effecting secure, authenticated communications across a network. Kerberos assumes it is in a distributed environment of unsecured workstations, moderately secure servers, and highly secure key-distribution machines. However, Kerberos is not a complete security solution as it does not handle authorisation for applications. Any determination of access authorisation and rights must be handled by other system services.

Kerberos uses private key encryption to provide three levels of protection. The lowest level only requires the user's authenticity to be established at the initiation of a connection, assuming that subsequent network messages flow from the authenticated principal. The next level up requires the authentication of each network message. The most secure level is where each message is encrypted as well as authenticated.

3.2.5 Transparency

A distributed system should try and hide to some degree, although maybe not completely, the differences between local and remote resources so that they are transparent to users. It is possible to identify various degrees of network transparency that can be required to achieve a distributed system. These include access, location, name, control, data, execution, and performance transparency. Transparency in a distributed system gives the advantages of easier development, support for incremental change, potential for increased reliability and a simpler user model of the distributed system.

3.2.6 Heterogeneous Environment

Heterogeneity in distributed systems is a major problem because it restricts the efficient and effective utilisation of resources. In general, the heterogeneity issue is a very important but still open problem which requires a global solution.

3.2.7 Scalability

A scaleable distributed system is one that can easily cope with the addition of users and sites, and whose growth involves minimal expense, performance degradation, and administrative complexity.

Scalability permeates almost every aspect of distributed system design. In centralised computer systems certain shared resources - memory, processors, input-output channels, are in limited supply and cannot be replicated indefinitely. In distributed systems the limitation in the supply of some of these resources is automatically removed but other limitations may remain as the design of the system does not recognise the need for scalability.

A distributed system should be designed so that the work involved in processing any single request to access a shared resource should be nearly independent of the size of the system. Techniques that have been successful in overcoming the problems associated with expanding systems include replicating data, caching, and using multiple servers.

3.2.8 Fault Tolerance

Sooner or later every man made system will fail because of physical faults. If the inherent fallibility of a system is not acceptable then redundancy must be provided in order to make the system fault-tolerant.

Therefore, an important issue in the design of a distributed programming model is the handling of processor failures. The usual approach for dealing with faults is to provide a mechanism for failure detection, and letting the programmer take care of clearing up. The main problem is to bring the system back to a consistent state; this is achieved by anticipating processor crashes and precautions, such as checkpoints, are taken during normal operation. To shield the programmer from these details, models have been suggested to make recovery from failures easier. Alternatively, a higher-level mechanisms for expressing which processes and data are important and how they should be recovered after a crash can be provided for the programmer (e.g. Argus, Aeolus). Current distributed systems support for fault tolerance ranges from specifically designed fault tolerant systems to systems with no special provisions what so ever.

Distributed systems also provide a high degree of availability in the case of faults. The availability of a system is a measure of the proportion of time that it is available for use. When a component in a distributed system fails only the work that was using that component is affected.

Networks are usually one of the weakest components in a distributed systems since they are not normally redundant. Failure of the network causes programs that use the network to hang until communication is restored, and the service to users will be interrupted. Overloading of the network degrades the performance and responsiveness of the system to users. Much effort has gone into the design of reliable and fault-tolerant networks.

Many different approaches have been taken in tackling fault tolerance. Amoeba uses a boot service for guarding important services. A process registered with the boot server is periodically polled to see if it is still alive, and if it fails to respond the boot server assumes that it has crashed and starts a new version of the process. The Clouds system supports fault-tolerant distributed applications through the use of persistent objects. Operations on objects can be grouped into atomic transactions, which either fully succeed, or fail completely with no side-effects. Resilient objects and atomic transactions together facilitate the implementation of fault-tolerant applications.

3.3 Systems for Distributed Computing

3.3.1 PVM - Parallel Virtual Machines

3.3.1.1 Introduction

PVM [Gei93] is a basic message passing system for heterogeneous collections of networked computers. Each computer may belong to any hardware classification that may be interconnected by a variety of networks, such as ethernet, etc. PVM support software executes on each machine in a user configurable pool, and presents a unified, general and powerful computational environment for concurrent applications. Access to PVM functions such as process initiation, message transmission and reception, and synchronisation is provided through PVM library routines. The execution location of specific application components is also under the optional control of the user, with the PVM system transparently handling message routing, data conversion where necessary and various other tasks that are necessary for operations in a heterogeneous, network environment.

3.3.1.2 PVM Overview

The goals of the original system (version 1.0) were to allow the execution of large parallel programs consisting of relatively independent components with good performance, on the best suited machine in a general, flexible, portable, and inexpensive concurrent programming environment, and also to provide a unified framework for developing large parallel systems efficiently. Many of the original ideas proposed were never implemented in a release version of PVM and this can cause some confusion. We will use PVM 1.0 to refer to the original experimental system.

PVM 3 provided many improvements and a change of emphasis. These changes included an updated user interface and a more stable and robust system, better multiprocessor integration, improved message passing capabilities and greater support for dynamic process groups. We now look at how PVM addresses the issues that arise in distributed computing.

3.3.1.2.1 Load Balancing Issues

Whereas PVM 1.0 used a form of dynamic load balancing PVM 3 has no capabilities in this area, and this is one of its most serious limitations. In version 3, hosts are assigned in a round-robin manner and specific hosts and architectures may be requested. However no attempt is made to obtain the relative speeds or loads of the hosts and load balancing is left to the programmer to implement.

3.3.1.2.2 Scalability Issues

PVM 3 is not really a scaleable distributed computing environment. In many instances communication is serialised through daemons (pvmds), while linear spawning and authentication's are used. This is workable for a small number of distributed computers but will eventually cause the system to bog down. Scalability is further impaired due to file descriptor limits on Unix; there's a limited number of TCP descriptors available at any one time.

3.3.1.2.3 Security Issues

Security is not a major concern in PVM which relies entirely on UNIX-based security techniques, and encourages the widespread use of .rhosts files to avoid explicit password authentication. The main aspects are that all processes run as user processes, virtual machines are set up on a per-user basis, slave pvmds are spawned with rsh() or rexec() and data encryption is not provided by PVM.

3.3.1.2.4 Fault Tolerant Behaviour

Fault tolerance must be implemented by the developer. No attempt is made to automatically recover tasks that are killed because of a host failure - PVM simply deletes the host from the virtual machine. No form of checkpointing is supported but processes can check the status of other processes.

3.3.1.2.5 Data Distribution/Locality Issues

Distributed shared memory was originally described in the PVM 1.0 documents but has not appeared since. This supported distributed locks, typed and untyped shared blocks and automatic data-format conversion applied to typed blocks. No support for data mapping, data distribution or data replication is provided at all.

3.3.1.2.6 Distributed I/O under PVM

PVM does not really address the issue of distributed input-output very much but does not inhibit the use of any other existing distributed I/O mechanism, e.g. NFS, AFS.

3.3.1.2.7 Multicomputer Interface

PVM 3 is designed so that the native multiprocessor calls can be compiled into the source. This allows the fast message-passing of a particular system to be utilised by the PVM application.

3.3.1.2.8 PVM Programming Environment

The main points of the PVM environment are that it has an easy to learn user interface, is portable with C and Fortran libraries, with a command line interface to the master pvmd.

There are some notable omissions, such as no applications or numerical libraries, no support for parallel debugging although several unsupported debugging features are provided, no way to trace execution other than debugging messages and no facility for monitoring performance.

Some of these short-comings can be overcome through various additions to the basic PVM system, such as the Paragraph tool which supports post-mortem traces and constructs a graphical view of execution from trace data.

3.3.1.3 The PVM Daemons

The PVM daemons are responsible for the majority of the inter-host communication occurring in the system, and also store most of the state needed by the system, such as the virtual machine configuration and client-process information. The master pvmd also provides a means for the user to access more information about the running system.

When a pvmd is started it spawns the slave daemons on all the hosts listed in the hostfile. Once all the PVM slave daemons have been initialised the master PVM daemon prints a message. If an error occurs while spawning a slave pvmd, a message describing the problem will be displayed.

The master PVM daemon can also be used in an interactive mode which allows the user to quit the program, get help, list the configuration of the virtual machine and to check and kill processes.

3.3.1.4 Functionality

PVM is portable to any machine running a version of UNIX that supports Berkeley sockets and provides basic message-passing facilities, barrier synchronisation, a form of user defined event handling and the ability to initiate new processes on demand. PVM is intended for medium to course grained parallel applications, which does not preclude large grain parallelism. PVM allows programmers to take advantage of the architectural advantages of different machines, so that each independent component of a program may be concurrently executed on the most suitable computer. Generally, the programming model used can either be SIMD/SPMD (a simple, general purpose client can be used to spawn functionally identical programs) or full MIMD/MPMD (e.g., a master/slave paradigm)[1]

3.3.1.5 Conclusion

PVM is being increasingly used around the world for distributed scientific computing. An important motivation for the use of PVM and other cluster computing systems is price performance as clusters are a lot more cost-effective than supercomputers for a given performance capability, for several classes of applications. Other motivations include a high degree of portability and a straightforward, robust interface that is well suited for scientific application development.

PVM's advantages are that it is small, simple, easy to run and understand, portable, with support for a heterogeneous environment and a large user base. Against this, PVM is an evolving research project, with no support for semaphores, message passing contexts, receipt selectivity, load balancing, execution tracing, or checkpointing and limited support for fault tolerance, scalability, and performance.

3.3.2 Message Passing Interface (MPI)

MPI [MPI93] is the new de facto standard for multicomputer and cluster message passing. The goal of MPI, simply stated, is to develop a widely used standard for writing message-passing programs.

Message passing is a paradigm used widely on certain classes of parallel machines, especially those with distributed memory. Although there are many variations, the basic concept of processes communicating through messages is well understood. Over the last ten years, substantial progress has been made in casting significant applications in this paradigm. In designing MPI the most attractive features of a number of existing message passing systems were used. Thus, MPI has been strongly influenced by work at the IBM T. J. Watson

[1] 'SPMD' stands for single program, multiple data; 'SIMD' indicates single program, multiple data; 'MPMD' stands for multiple program, multiple data; 'MIMD' indicates multiple program, multiple data.

Research Centre, Intel's NX/2, Express, nCUBE's Vertex, p4 [But92], and PARMACS [Cal94], along with contributions from Zipcode [Skj93b], PVM [Gei93], and PICL [Gei92].

The main advantages of establishing a message passing standard are portability, ease-of-use and support for parallel libraries and the creation of large-scale multicomputer software. In a distributed memory communications environment in which the higher level routines and/or abstractions are built upon lower level message passing routines the benefits of standardisation are particularly apparent. Furthermore, the definition of a message passing standard provides vendors with a clearly defined base set of routines that they can implement efficiently, or in some cases provide hardware support for, thereby enhancing scalability.

MPI provides a higher level of abstraction than common message-passing systems, which helps library writers develop more efficient and understandable code. MPI was designed to support primitives that would map efficiently to the existing and expected high performance message passing hardware of multicomputers and gigabit cluster networks while allowing portable MPI libraries to be developed.

Parallel libraries are needed to hide the intricate technical details associated with distributed implementations of important algorithms. Libraries help ensure consistency in program correctness and quality of implementation, while offering a higher level of portability than a message passing system, such as MPI, can provide. In the distributed memory message-passing environment it is difficult to write libraries with either vendor or portability systems. In order to develop and support such libraries, three key areas must be addressed:

1. A *safe communication space* (isolation) which guarantees that a library can send and receive point-to-point messages without interference from other point-to-point messages generated in the system.

2. Support for *collective operations*, not just point-to-point.

3. *Abstract names for processes* based on virtual topologies which do not rely on hardware-dependent names.

MPI supports these areas through the following features:

1. *Process Groups* define an ordered collection of processes, each with a rank. Process groups define the low-level names, rank of sender and receiver, and the set of participants for inter-process communication, point-to-point message passing, and collective communication respectively.

2. *Virtual Topologies* provide an alternative to simple rank naming for libraries with support for both Cartesian and general graph topologies as well as user-definable topology strategies.

3. *Contexts* allow separate safe "universes" of message-passing for process groups in MPI and help restrict the scope of messages. Contexts in MPI are implemented with communicators.

4. *Communicators* hide contexts, groups and virtual topologies in an object that provides the appropriate scope for all communication operations in MPI. Processes and contexts are bound together by communicators to form a safe communication space within the group.

Some possible MPI-2 extensions which were proposed at the February 1994 MPI Forum meeting include active messages, I/O, process start-up, dynamic process control, remote store/access, Fortran 90 and C++ language bindings, graphics and real-time support.

3.3.3 OSF's Distributed Computing Environment

To meet the needs of distributed computing the Open Software Foundation (OSF) have put forward it's Distributed Computing Environment (DCE). OSF's DCE is an integrated set of operating system and network independent services that support the development, use and maintenance of distributed applications [Byt94].

DCE is constructed in layers, from the basic services (e.g., operating systems) up to higher-end clients of services (e.g., applications). Security and management are essential to all layers in the model. Currently DCE consists of seven tools and services that are divided into fundamental distributed services and data-sharing services.

The fundamental distributed services include threads, RPCs, directory services, time services, and security services. Data Sharing Services build on top of the fundamental services and include Distributed File System (DFS) and diskless support. The OSF has reserved space for possible future services, such as spooling, transaction services, and distributed object-oriented environments.

3.3.3.1 Threads

Threads were introduced to allow multiple related tasks to co-exist and execute in a single process and are an important emerging model for expressing parallelism within a process, especially within a distributed environment. The DCE RPC, security, directory, time services, and DFS (Distributed File System) all use the threads service.

3.3.3.2 Remote Procedure Calls (RPC)

A well known and tested method for implementing a distributed application is the RPC (Remote Procedure Call), which extends a function call across networks. RPCs handle the core responsibilities of the distribution such as the semantics of the call, binding to the server, and communication failure transparently, simply, efficiently, and reliably. This allows the programmer to concentrate on the distribution of the problem at hand. The RPC protocol is consistent and clearly specified and is not subject to user modification. The DCE RPC fits in well with the other needs of DCE such as naming, DFS, security, and time service.

3.3.3.3 Distributed Directory Service (Naming)

The directory service is used to find users, resources, data, applications, etc., in a distributed network. Naming or directory services must map large numbers of system objects to user-oriented names in a transparent manner. OSF specified a two-tiered architecture for the name service to address both intracell and world-wide communications. Cells are the fundamental

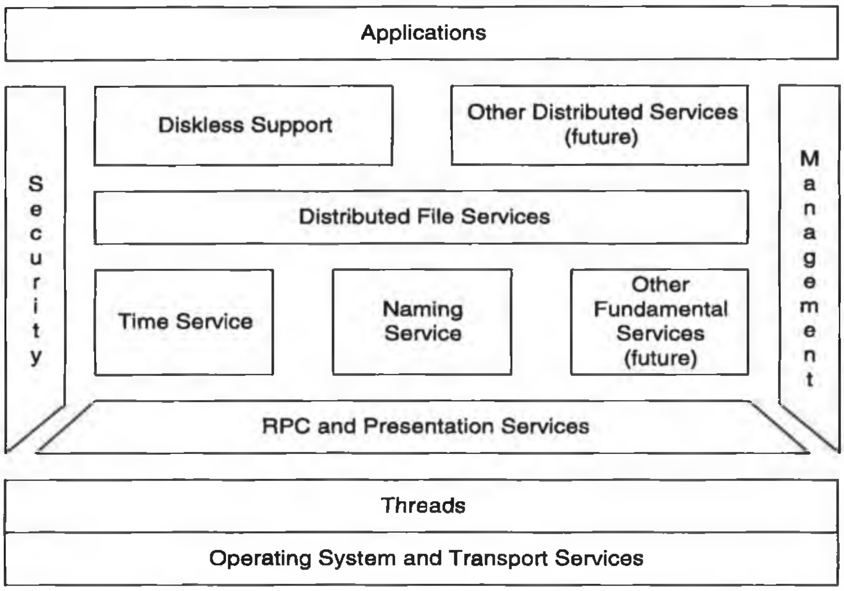

Applications
Diskless Support
Other Distributed Services (future)
Security
Management
Distributed File Services
Time Service
Naming Service
Other Fundamental Services (future)
RPC and Presentation Services
Threads
Operating System and Transport Services

Figure 8: The OSF DCE Architecture

The DCE architecture follows a layered model. The most basic, or supplier, services are at the bottom, with the highest level being service consumers. The security and management services apply to all layers organisational unit and can map to social or organisational boundaries etc., consisting of computers that must communicate frequently with one another.

3.3.3.4 Distributed Security Service

There are two general categories of security; authentication and authorisation. Authentication verifies the identity of an entity while authorisation, which is not enough on its own, is responsible for checking and granting access rights to resources. OSF security is based on the Kerberos authentication system, augmented by security components such as a registry and authorisation service and is also looking at further improvements. DCE applications will also be portable from Kerberos to public-key authentication schemes such as that provided by RSA.

3.3.3.5 Distributed File System (DFS)

The OSF DFS, which is based on the Andrew File System (AFS), is intended to provide transparent access to any file residing anywhere on a network. A major concern for such a system is that it remain simple and easy to use while providing a uniform name space. Other considerations are integrated security, data consistency and availability, reliability and recovery, performance and scalability to very large configurations without performance degradation, and coherent, location-independent management and administration.

3.3.3.6 Distributed Time Service (DTS)

Many distributed services, such as distributed file systems and authentication services, compare dates generated on different computers. For the comparison to be meaningful, DCE must support a consistent time stamp. DCE uses a time server to provides the time to other systems for the purpose of synchronisation, and is interoperable with the NTP, the protocol used by the Internet. There are three types of time servers (local, global and courier) to co-ordinate network time. At periodic intervals, servers synchronise with every other local server on the LAN via the DTS protocol. Any non-time server system is called a clerk.

3.3.3.7 Areas for Extension

Object orientation will prove fundamental to the rapid proliferation of network based applications, for some of the same reasons that are propelling the transparency of DCE to developers. Currently it is too hard to write a network based application without either extensive training or a technology that camouflages the intricacies of the network. Object orientation provides this necessary transparency. OSF is exploring extensions to its RPC interface definition language that will add object oriented functionality. Once implemented, such features will support the Object Management Group's CORBA (Common Object Request Broker Architecture) on top of the DCE infrastructure.

3.3.4 A Brief Note on Other Systems

Those interested in other systems should investigate p4 [But92], Parmacs [Cal94], and Zipcode [Skj93b] and the discussion of Linda in section 2.4.4.

3.4 Distributed Objects

Distributed object computing is rapidly becoming the next computer industry battleground, superseding the operating system battle. Everyone working in modern enterprise will be engulfed because distributed computing represents the direction in which the broad mainstream of information technology will likely evolve. At the heart of distributed computing are interoperable objects that go beyond the traditional boundaries imposed by programming languages, process address space, or network interfaces.

How is the idea of distributed objects different from the basic ideas of distributed processing? Objects are an enabling technology for distributed systems. In the case of distributed systems, the concerns - naming, address-space conversion, transport protocols, interface descriptions, etc., are many. Objects provide a tractable means of organising the complexity of a modern operating system, thereby simplifying distributed processing, by combining the data and process together.

3.4.1 CORBA

The model for client/server is maturing into one based on peer-to-peer distributed processing. Consequently a considerable amount of attention is being focused on technologies for linking applications and objects across machine boundaries in a heterogeneous, networked environment. These technologies fall under the title of 'distributed object computing' [Dob94].

Many of the technologies vying for dominance rely on an emerging standard called the "Common Object Request Broker Architecture" (CORBA) specification. Current systems based on or compliant with CORBA include the Distributed System Object Model (DSOM) from IBM, Digital's Object Request Broker (ORB), Portable Distributed Objects (PDO) from Next, and Sunsoft's Distributed Objects Environment (DOE).

The CORBA specification is being promoted by the Object Management Group (OMG), which is a consortium of three hundred companies and was founded in 1989. The CORBA specification defines the architecture of an ORB whose job is to enable and regulate interoperability between objects and applications. This facility is part of a larger vision called the "Object Management Architecture" (OMA), which defines the OMG object model.

The concern of the CORBA specification is solely the interaction of applications and objects and the mechanisms that enable it. The vision of the OMG is clearly cross platform and cross operating system. It is a standard description of an architecture but it is not a standard for implementation of that architecture, and it is not as well defined as it needs to be.

3.4.2 SOM & Distributed SOM

IBM's System Object Model (SOM) [Dob94a] was designed to overcome several major obstacles to the widespread use of object-class libraries. The goal of SOM is to enable the development of 'system objects' which can be distributed and subclassed in binary form, used across languages, and upwardly binary compatibility - i.e. subsequent modifications are possible without having to recompile.

SOM supports all the concepts and mechanisms normally associated with object-oriented systems, including inheritance, encapsulation, and polymorphism, as well as a number of advanced types of method dispatch.

3.4.2.1 The SOM Framework and Distributed SOM

SOM is a peer-to-peer communications vehicle interconnecting objects and frameworks with each other, rather than to a central 'master' controller. SOM includes with it a number of frameworks, which are interrelated sets of SOM objects designed to solve a particular problem, such as object persistence, object replication, and object distribution.

The purpose of the distribution framework (called 'distributed SOM' or DSOM) is to seamlessly extend SOM's internal method-dispatch mechanism so that methods can be invoked in a programmer transparent way on objects in a different address space or in a different machine from the caller.

Components that are included in DSOM allow messaging between objects in different address spaces on the same machine. Other components, such as marshalling, transport, naming or security, can be added to support messaging between objects on different machines. Components can also be replaced, depending upon the particular distributed-computing environment that needs to be supported.

3.4.2.2 SOM, DSOM, and CORBA

IBM is trying to achieve many of the same objectives that the Object Management Group (OMG) is trying for with its CORBA specification. Their common goal is to facilitate the interoperability of objects independent of where they are located, the programming language in which they are implemented, or the operating system or hardware architecture on which they are running.

The DSOM class library is fully compliant with the CORBA specification, supporting all CORBA data types, functionality, and programming interfaces. DSOM is a framework built using the SOM technology that allows developers to construct distributed-object applications.

SOM can be looked on as a single-address space, object request broker (ORB) that provides interlanguage interoperability and supports binary subclassing and upward binary compatibility.

SOM deals with object implementation rather than the narrower focus of the CORBA specification which defines objects without regard to implementation. SOM advances on CORBA by supporting implementation inheritance and polymorphism, providing metaclasses that are manipulated as first order objects, and allowing dynamic addition of methods to a class interface at run-time.

3.5 Conclusions

PVM and Linda, among others, have evolved over the past few years, but none of them can be considered fully mature [Don93]. The field of network based concurrent computing is relatively young and research on various aspects is ongoing. Although basic infrastructures have been developed many of the refinements that are necessary are still evolving.

The rapid increases in computing power presents many challenges for network computing systems. One aspect concerns scaling to hundreds and perhaps thousands of independent machines. Protocols to support scaling and other system issues are currently under investigation. Under the right conditions, the network based approach can be effective in coupling several similar multiprocessors, resulting in a configuration that might be economically and technically difficult to achieve with hardware.

Applications with large execution times will benefit greatly from mechanisms that make them resilient to failures. Currently few platforms support application level fault tolerances. In a network based computing environment application resilience to failures can be supported without specialised enhancements to hardware or operating systems. Approaches based on checkpointing, shadow execution and process migration are being looked at as possible strategies for enabling fault tolerant applications.

The performance and effectiveness of network based concurrent computing environments depends largely on the efficiency of the support software and on the minimisation of overheads.

Another issue to be addressed concerns data conversions that are necessary in networked heterogeneous systems. Aside from byte ordering, there is also a need to handle differences in wordsize and precision, when operating in a heterogeneous environment.

Another concern is the efficient implementation of distributed primitives which are inherently expensive parallel computing operations, such as barrier synchronisation, polling, distributed fetch-and-add, global operations, automatic data-decomposition and distribution, and mutual exclusion. This is all the more important in an irregular environment (where interconnections within hardware multiprocessors are much faster than network channels), as such operations can cause bottlenecks and severe load imbalance.

Hopefully the object oriented approach can go some way towards hiding many of these issues and concerns from the application developer while providing a powerful means of harnessing the full potential of distributed computing systems.

3.6 References

[Ack82] Ackerman, W.B., "Data Flow Languages," Computer, 15. 1982.

[Alm85] Almes, G.T., Black, A.P., Lazowska, E.D., Noe, J.D., "The Eden System: A Technical Review," IEEE Trans. Software Eng. Vol. SE-11, pp 3-59. Jan. 85.

[Ame87] America, P., "POOL," Object-Oriented Concurrent programming, A. Yonezawa (ed.), MIT Press. 1987.

[Art86] Artsy, Y., Chang, H.-Y., Finkel, R., "Process Migrate in Charlotte," Computer Science Technical Report No. 655, University of Wisconsin-Madison. 1986.

[Ath88] Athas, W.C., Seitz, C.L., "Multicomputers: Message-Passing Concurrent Computers," IEEE Computer, Vol. 21, No. 8, pp. 9-24. Aug.1988.

[Bal90] Bal, H., "Programming Distributed Systems," Silicon Press, Prentice Hall, ISBN 0-13-722083-9.

[Ber86] Berglund, E.J., "An Introduction to the V-system," IEEE Micro, Vol. 6, No. 4, pp. 35-52. Aug. 1986.

[Bir84] Birrell, A.D., Nelson, B.J., "Implementing Remote Procedure Calls," ACM Transactions on Computer Systems, Vol. 2, No. 1, pp. 39-59. Feb. 1984.

[But92] Butler, R., Lusk, E., "User's Guide to the P4 Programming System," Argonne National Laboratory, Technical Report TM-ANL-92/17. 1992.

[Byt94] Byte, Special Report on Distributed Computing, pp. 125, June 1994.

[Cal94] Calkin, R., Hempel, R., Hoppe, H., Wypior, P., "Portable programming with the parmacs message-passing library," Parallel Computing. 1994.

[Dob94] Dr. Dobbs, Special Report, "Interoperable Objects," Winter 94/95, "OMG's CORBA," pp 8-12; and Dr. Dobbs, "Interoperable Objects," Oct. 94, pp 18-39.

[Dob94a] Dr. Dobbs, Special Report, "Interoperable Objects," Winter 94/95, "IBM's System Object Model," pp 24-28.

[Don93] Dongarra, Jack, Geist, G.A., Manchek, R., Sunderam, V.S., "Integrated PVM Framework Supports Heterogeneous Network Computing," ORNL TR available in /pvm/comp-phy.ps

[Geh87] Gehani, N.H., "Message Passing: Synchronous versus Asynchronous," AT&T Bell Laboratories, Murray Hill, NJ. 1987.

[Gei92] Geist, G., Heath, M., Peyton, B., Worley, P., "A User's Guide to PICL: a Portable Instrumented Communication Library," Technical Report ORNL/TM-11616, Oak Ridge National Laboratory. May 1992.

[Gei93] Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., Sunderam, V., "PVM 3 Users's Guide and Reference Manual," Technical Report ORNL/TM-12187, oak Ridge National Laboratory. May 1993.

[Gos92] Goscinski, Andrzej, "Distributed Operating Systems - The Logical Design,"Addison - Wesley, 1992. ISBN 0 201 41704 9.

[Hoa78] Hoare, C.A.R., "Communicating Sequential Processes," Englewood Cliffs, NJ: Prentice-Hall. 1985.

[Hog84] Hogger, C.J., "Introduction to Logic Programming," Academic Press,1984

[May84] May, D., Shepherd, R., "The Transputer Implementation of Occam," Proc Intl. Conf. of Fifth Generation Computer Systems 1984, pp. 533-541, Tokyo. Nov 1984.

[MPI93] Message Passing Interface Forum. Document for a Standard Message-Passing Interface. Technical Report No. CS-93-214., University of Tennessee, Nov. 1993.

Send a mail message to netlib@ornl.gov containing the line send index from MPI, or use anonymous ftp from netlib2.cs.utk.edu.

[Nee82] Needham, R.M., Herbert, A.J., "The Cambridge Distributed Operating System," Addison-Wesley. 1982.

[Nel81] Nelson, B.J., "Remote Procedure Call," Report CMU-CS-81-119, Carnegie-Mellon University, Pittsburgh, PA. May 1981.

[Pow83] Powell, M.L., Miller, B.P., "Process Migration in DEMOS/MP," Proc. 9th Symp. Operating Systems Principles, pp. 110-119, ACM SIGOPS, Bretton Woods, NH. Oct. 1983.

[Ran88] Ranka, S., Won, Y., Sahni, S., "Programming a Hypercube Multicomputer," IEEE Software, Vol. 5, No. 5, pp.. 69-77. Sep. 1988.

[Skj93a] Anthony Skjellum, "The Multicomputer Toolbox: Current and Future Directions," In Anthony Skjellum and Donna S. Reese, editors, Proceedings of the Scalable Parallel Libraries Conference, IEEE Computer Society Press. Oct. 1993.

[Skj93b] Skjellum, A., et al., "The Design and Evolution of Zipcode," Anonymous FTP: get.internet.site. November 12, 1993.

[Tan85] Tanenbaum, A.S., van Renesse, R., "Distributed Operating Systems," ACM Computing Surveys, Vol.17, No. 4, pp. 419-470. Dec. 1985.

[Tan88] Tanenbaum, A.S., van Renesse, R., "A Critique of the Remote Procedure Call Paradigm," Proc. of the EUTECO 88 Conf., pp 775-783, ed. R. Speth, North-Holland, Vienna, Austria. April 1988.

[The85] Theimer, M., Lantz, K., Cheriton, D.R., "Pre-emptable Remote Execution Facilities for the V-system," Proc. 10th Symp. Operating Systems Principles, pp. 2-12, ACM SIGOPS, Orcas Island, WA. Dec. 1985.

[Wal83] Walker, B., et al, "The LOCUS Distributed Operating System," Proc. 9th ACM Symp. on Operating System Principles, Bretton Woods, New Hampshire, pp. 49-70. 1983.

4. DISTPROC
A Distributed Processing System

4.1 Introduction

DISTPROC is a basic distributed processing system for collections of networked computers and presents a unified and powerful computational environment for concurrent applications.

DISTPROC is suitable for concurrent applications composed of many independent parts. DISTPROC falls into the generic distributed computing category where workstations are connected via a network (e.g. ethernet), rather than into the high performance distributed computing category where supercomputers and very high speed connections are used. As a loosely coupled computing environment DISTPROC is a viable computing system and has been used on the factorisation of very large numbers.

Access to the DISTPROC services is provided through the DISTPROC Client library which transparently handles the various tasks that are necessary for operations such as process creation, termination and synchronisation.

The DISTPROC System was implemented, as part of the M.Sc., in C++ and the source code is presented in Chapter 8.

4.2 Design Objectives of the DISTPROC System

The DISTPROC System was designed and implemented around the following goals. It should:

- use existing resources,
- provide good performance,
- provide a general, flexible, portable and inexpensive concurrent programming environment,
- be a stable and robust system,
- be easy to program and use,
- help investigate the issues surrounding distributed systems,
- allow the execution of large parallel applications consisting of relatively independent components,

and most importantly that it should

- be capable of dealing with the factorisation of very large numbers.

4.3 Distributed Systems Issues

Before proceeding further a brief discussion on how the DISTPROC System addresses the various issues that arise in distributed computing is presented.

4.3.1 Distributed Scheduling

Distributed scheduling in DISTPROC is based on a prioritised round robin scheduler. Currently DISTPROC knows nothing about the relative speeds or loads of the hosts. One of the most serious limitations is the lack of dynamic load balancing in DISTPROC.

4.3.2 Scalability Issues

DISTPROC is not really a scaleable distributed computing environment. For instance, communication is serialised through a DISTPROC Server, which is workable for a small number of distributed computers but will eventually cause the system to bog down. Scalability is also affected by operating system and network limitations on which the system is built.

4.3.3 Security Issues

Specific support for security is not provided by the DISTPROC System, but it does support Unix based security techniques and encourages the use of .rhosts files to avoid explicit password authentication. DISTPROC can run in either multi-user or single user mode. In the single user mode the system is running solely for that user so security may be relaxed. Regardless, all Remote Processes run as user processes and are spawned with rsh(). No mechanism for data encryption is provided.

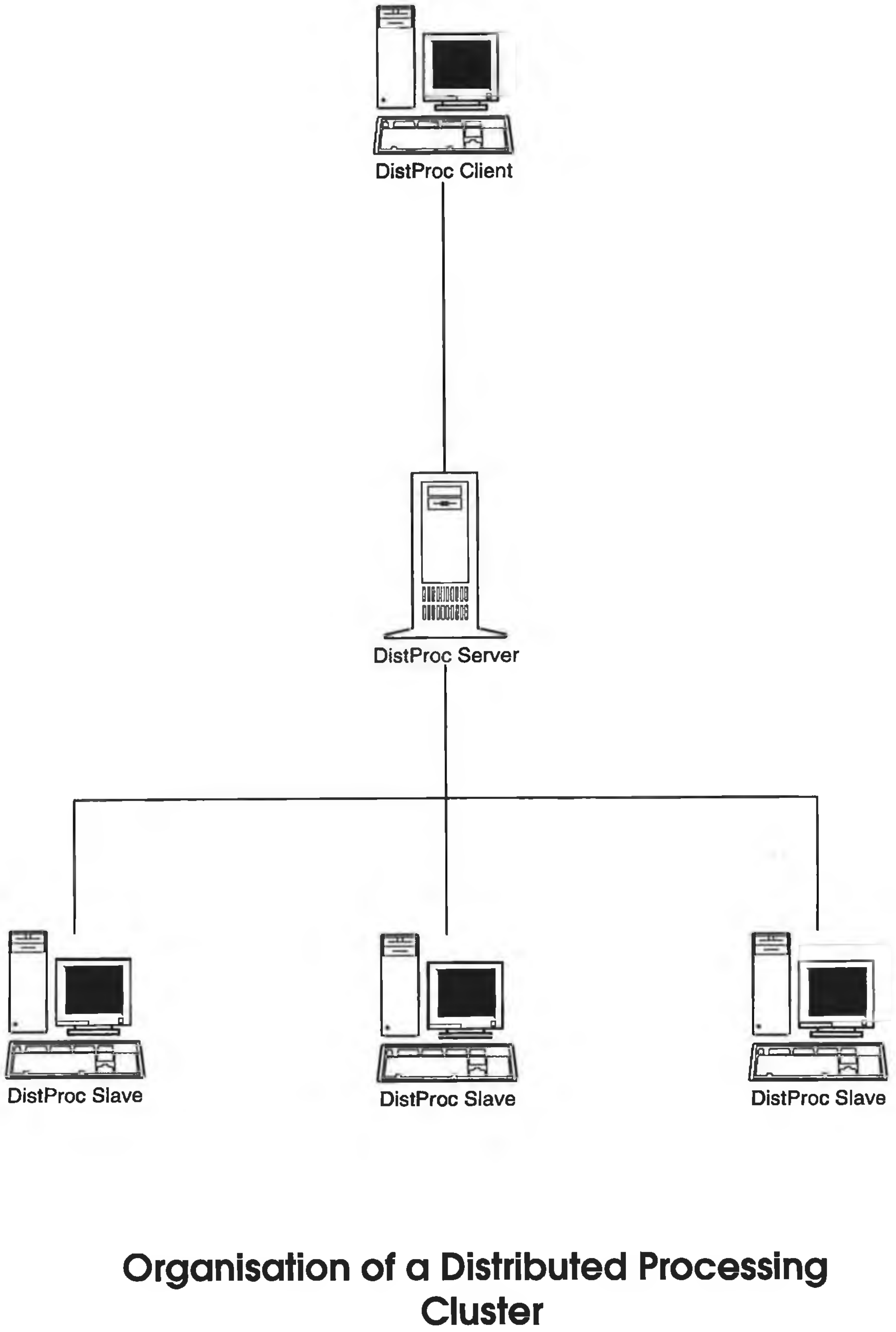


Figure 9. Organisation of a Distributed Processing Cluster

4.3.4 Fault Tolerant Behaviour

There is a certain degree of fault tolerance in the DISTPROC System. Recovery from situations where DPSlaves unexpectedly stop for whatever reason is possible. The DPSlave is removed from the DPServers tables and any uncompleted RemProcs assigned to it are redistributed if possible.

The DPServer is a weak link in the system as it is central to running of the system. No form of checkpointing is supported but could be added to the system. Further support for fault tolerance could be implemented by the user.

4.3.5 DISTPROC Programming Environment

The DISTPROC environment is portable and has a simple and easy to learn interface. However, it is lacking in support for parallel debugging and execution tracing and performance monitoring.

DISTPROC is portable to any machine running a version of Unix that supports Berkeley sockets and is intended for large grained parallel applications. DISTPROC is designed to allow programmers to take advantage of multiple machines, so that each independent component of a program may be concurrently executed on separate computers. Generally, the programming model used is full MIMD/MPMD (e.g., a master/slave paradigm) but can also be SIMD/SPMD (a simple, general purpose client can be used to spawn functionally identical programs).

4.4 Design and Implementation

4.4.1 Overall System Architecture

The DISTPROC System consists of three separate, yet integrated, components; the DISTPROC Server, Slaves, and Clients. The linch-pin of the system is the DISTPROC Server, which is responsible for co-ordinating and managing the whole system. The DISTPROC Server stores most of the state needed by the system, such as the virtual machine configuration and Client Remote Process information, and also provides a means for the user to access more information about the running system. The DISTPROC Slaves are the workhorses of the system that execute Remote Processes on behalf of a DISTPROC Server. As each DISTPROC Slave is initialised the DISTPROC Server daemon prints a message, and if an error occurs while spawning a DISTPROC Slave, a message describing the problem will be displayed. Access to the system is through the DISTPROC Client library, which provides a controlled interface to the DISTPROC Server.

The various elements of the DISTPROC System are distributed across a local area network, and communicate by means of a messaging system based on persistent objects and uses the TCP/IP communications protocol.

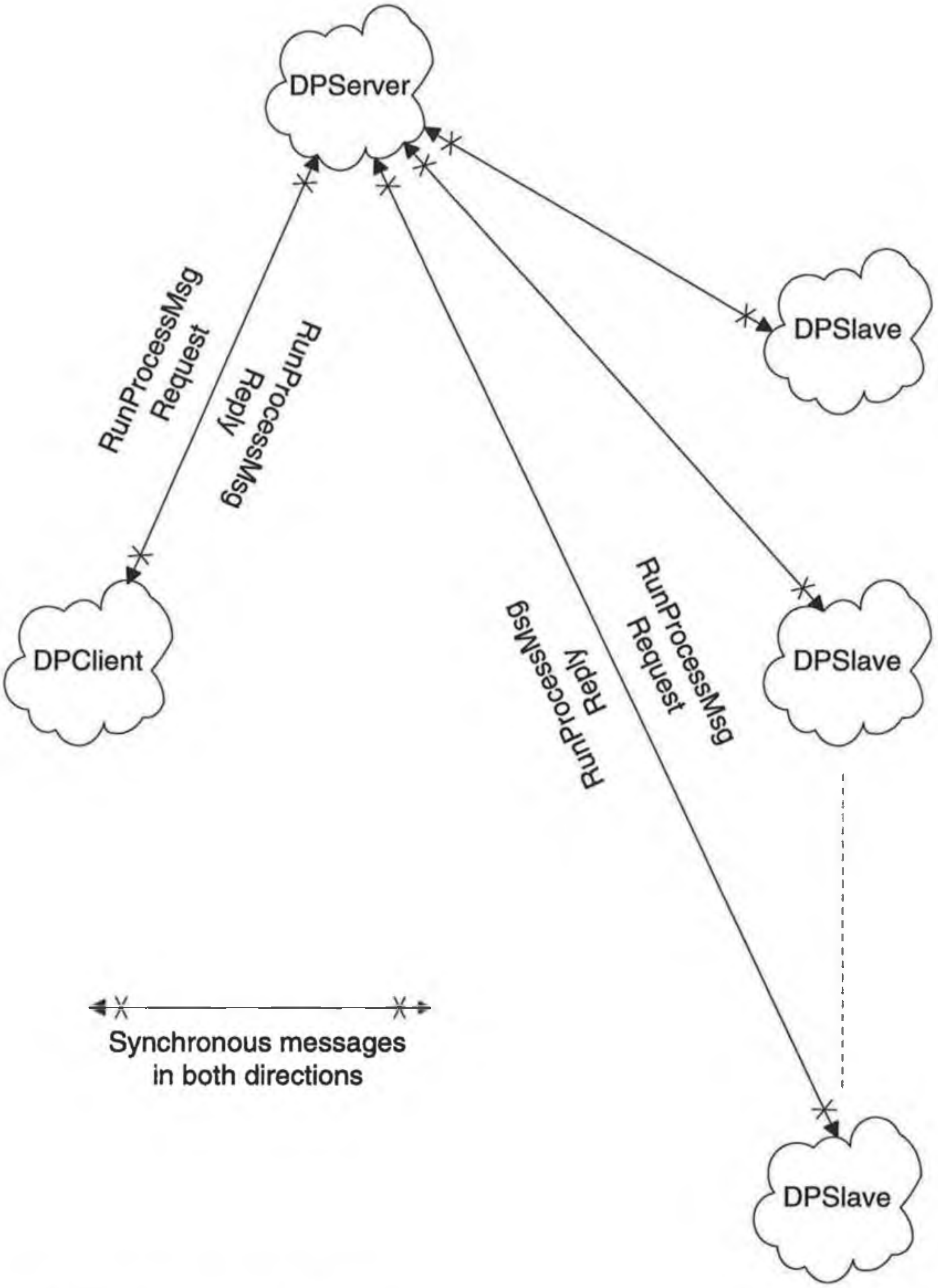


Figure 10. Overview of the DISTPROC System

4.4.1.1 Messaging System

The messaging system consists of eleven distinct messages. Each message in the system is an instance of a message class and every system message is derived from the BaseMsg class. The BaseMsg class provides information on the message type, size, source, and destination, etc., as well as the necessary data members.

The messaging passing system in the DISTPROC system is built around the idea of persistence. Every message in the system must be able to save and reload an instance of itself.

To send a message it must be packed (marshalled, saved) into a buffer before transmission, which is achieved through persistence. Every message class must provide two member functions - SAVE and LOAD - which save and restore an object's state from a buffer. Once a message is saved, it can be transmitted to a DISTPROC Server, Slave or Client.

Since all system messages are derived from the BaseMsg class, and the BaseMsg is packed first, the first few bytes of a buffer are always known, thereby allowing messages to be easily identified.

The following messages (events) are used throughout the system, between Clients, Servers and Slaves. The meaning of each message remains the same across the system, however the systems re-action to a message depends on the context in which it occurs, i.e. between a Server and a Slave or between a Client and a Server. Messages with the same name may result in different actions depending on the context.

There are two problems in loosely-coupled distributed systems, namely: unpredictable delays in communications mechanism, and the possibility of partial failure. The first problem is well catered for already, any further 'layers' will add too much communications overhead. The partial failure problem is a lot harder to solve.

Message	Usage	Client ↔ Server	Server ↔ Slave
BaseMsg	Base message from which all others are derived.	N/A	N/A
ServerShutDownMsg	Sent by a Server to Slaves and Clients before shutting down.	X	X
ClientConnectMsg	Enrols the Client application with a Server.	X	
ClientDisconnectMsg	Unrolls the Client application from a Server.	X	
SlaveConnectMsg	Enrols a slave with a Server.		X
SlaveDisconnectMsg	Unrolls a slave from a Server.		X
RunProcessMsg	Starts the execution of a Remote Process.	X	X
WaitProcessMsg	Waits for a Remote Process to finish execution..	X	
KillProcessMsg	Used by Clients and Servers to stop the execution of a Remote Process.	X	X
ResultMsg	Used by Remote Processes to send results to a Server.		X
DOCMsg	Sent by a slave to a Server upon the abnormal termination of a child process (Remote Process).		X
QueryStatusMsg	Used to query a Server about various resources.	X	X

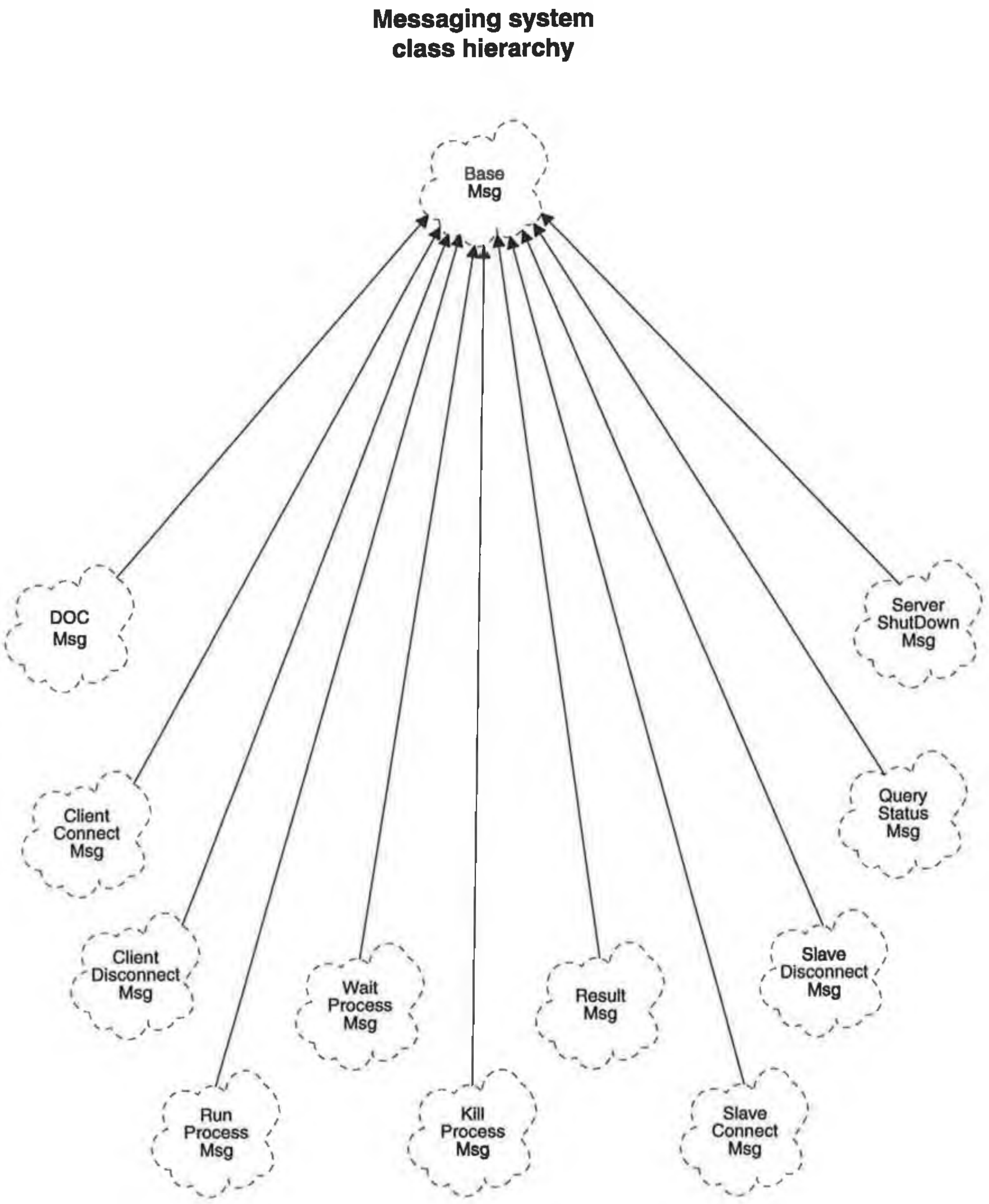
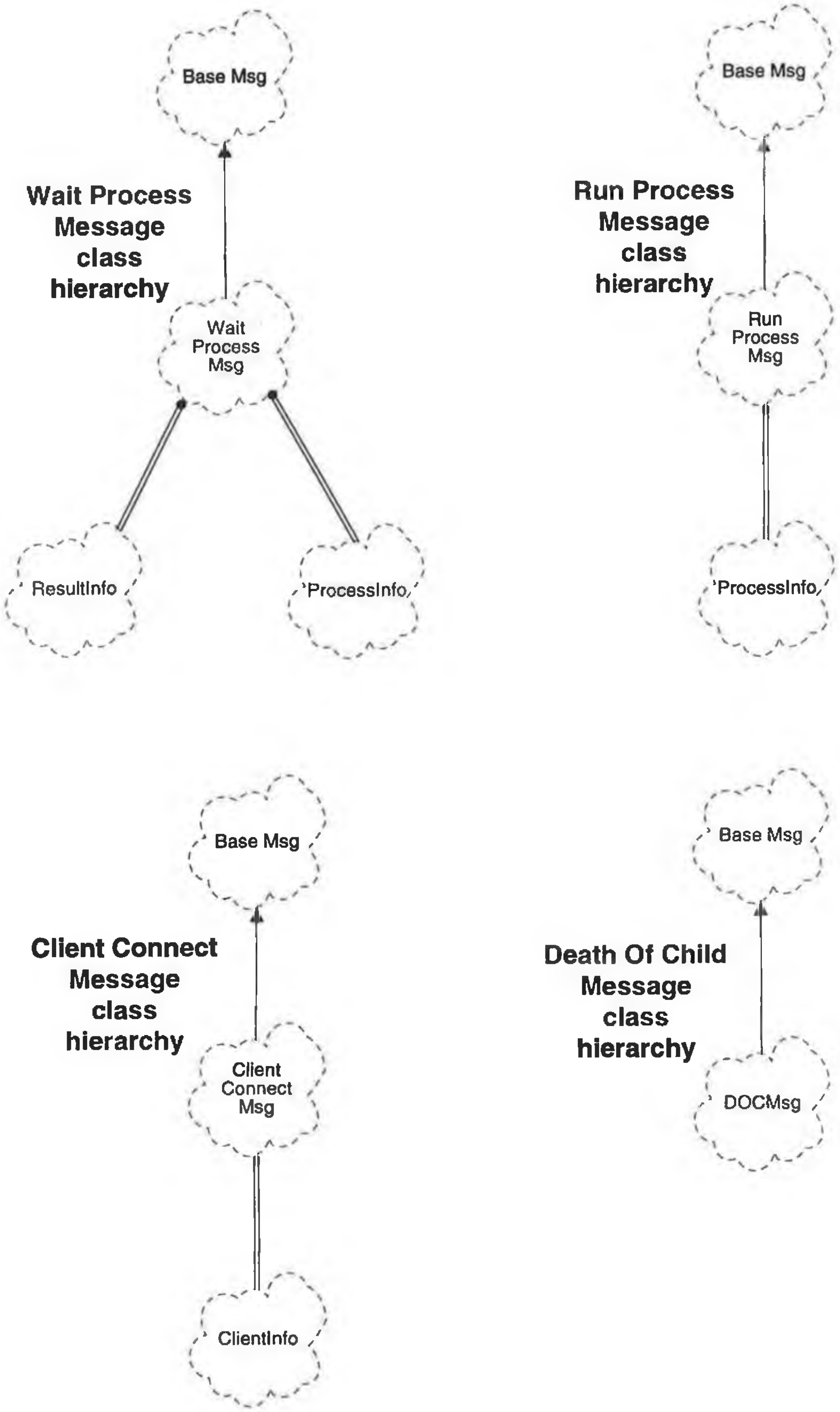


Figure 11. Messaging System Class Hierarchy


Base Msg
Wait Process Message class hierarchy
Wait Process Msg
ResultInfo
ProcessInfo
Base Msg
Run Process Message class hierarchy
Run Process Msg
ProcessInfo
Base Msg
Client Connect Message class hierarchy
Client Connect Msg
ClientInfo
Base Msg
Death Of Child Message class hierarchy
DOCMsg

4.4.1.2 Figure 12. Various Message Classes Remote Processes

Remote Processes in the DISTPROC system can have one of the following states:

Status	***Meaning***
PROCESS_ALLOCATED	A Remote Process is allocated, but is not yet running.
PROCESS_RUNNING	A Remote Process is currently executing on a slave.
PROCESS_FINISHED	A Remote Process finished successfully and a result is ready.
PROCESS_ABORTED	A Remote Process finished unsuccessfully and no result is available, e.g. core dump, killed, et cetera.

Remote Processes are started on a DISTPROC Slave through a call to the rsh() system call, which in conjunction with .rhost files will handle security - if desired. All Remote Processes run at reduced priority and therefore do not impinge excessively on other users.

.rhosts files are used to allow DISTPROC Slaves to execute a Remote Process on behalf of a Client. This adds the overhead of the Unix security mechanism, and may not always be necessary. If so, the DISTPROC System can be setup without the security overhead, by running the system and all Remote Processes under the same account.

Should Remote Processes send results to the DISTPROC Server directly, or via the relevant Slave? Sending a result directly to the Server is faster and more direct, but means that Remote Processes need to know more information about the Server and is yet another point of access to the Server. Sending results back via a Slave is slower, indirect, and requires that Remote Processes know about it's Slave/parent process, but has the benefit of reducing the number of access points to a DISTPROC Server.

In the DISTPROC System, Remote Processes return results directly to Servers and then exit. DISTPROC Slaves catch the resulting DOC (Death Of Child) signal, check the status, and only if there was an error will a DOCMsg be sent to the appropriate DISTPROC Server. On receipt of a DOCMsg, a DISTPROC Server first checks if it has received a result from the relevant RemProc; if so there is no problem. However if no result has been received the aborted RemProc must be marked as PROCESS_ABORTED in the Process System Table and either re-started on another DISTPROC Slave or reported as aborted when the relevant DISTPROC Client tries to wait (for a result) on the process.

4.4.1.3 How a Server and a Slave Co-operate to Create a Remote Process

When a DISTPROC Server receives a message from a DISTPROC Client requesting it to execute a Remote Process the operations outlined in table below are carried out by and between a DISTPROC Server and a DISTPROC Slave:

On the Server	*On the Slave*
• Select which Slave to use.	
• Send a RunProcessMsg to Slave.	
	• Receive a RunProcessMsg.
	• Spawn new child process.
	• Send input parameters to child process.
	• Send return code and process info to Server.
• Receive information on new process and update tables.	

Each incarnation of a DISTPROC Slave's run time system supports dynamic creation of local processes. A Remote Process is created by sending a message to the run time system on the slave computer, asking it to create a local process. The implementation of the run time system maps DISTPROC processes (Remote Processes) onto operating system processes.

4.4.1.4 How the System Deals with Results

When a Remote Process has completed its calculations it must return a result to its Client application. This is carried out as follows.

To return a result a Remote Process first packs the results into a ResultMsg, connects to its DISTPROC Server, and sends the completed message. On receipt of a ResultMsg a DISTPROC Server carries out one of the following:

- If there are any outstanding wait requests for the newly returned result then it is passed straight on to the appropriate Client. The Process, Slave, and Wait System Tables must also be updated to reflect the fact that process has finished running. Otherwise,

- The newly returned result is saved in the Result System Table and status of the Remote Process that generated the result is changed to PROCESS_FINISHED in the Process System Table. The appropriate Slave must also be updated in the Slave System Table to reflect the fact that it there is one less process running.

4.4.1.5 Fault Tolerance

In the DISTPROC System no message passing system is made publicly available and it is not intended that inter process communication be part of a DISTPROC Application. Therefore, no inter-dependence should exist among Remote Processes and if a Remote Process abnormally dies, it can be restarted without worrying about the state of, and the effect on, other Remote Processes.

In the system, the DISTPROC Server is critical, and if it goes down the system comes to a halt and cannot recover. Once this happens all connected DISTPROC Slaves shutdown and any connected DISTPROC Clients receive an error if a request is made to the downed DISTPROC Server.

The system can also detect and handle the abnormal termination of Remote Processes and DPSlaves.

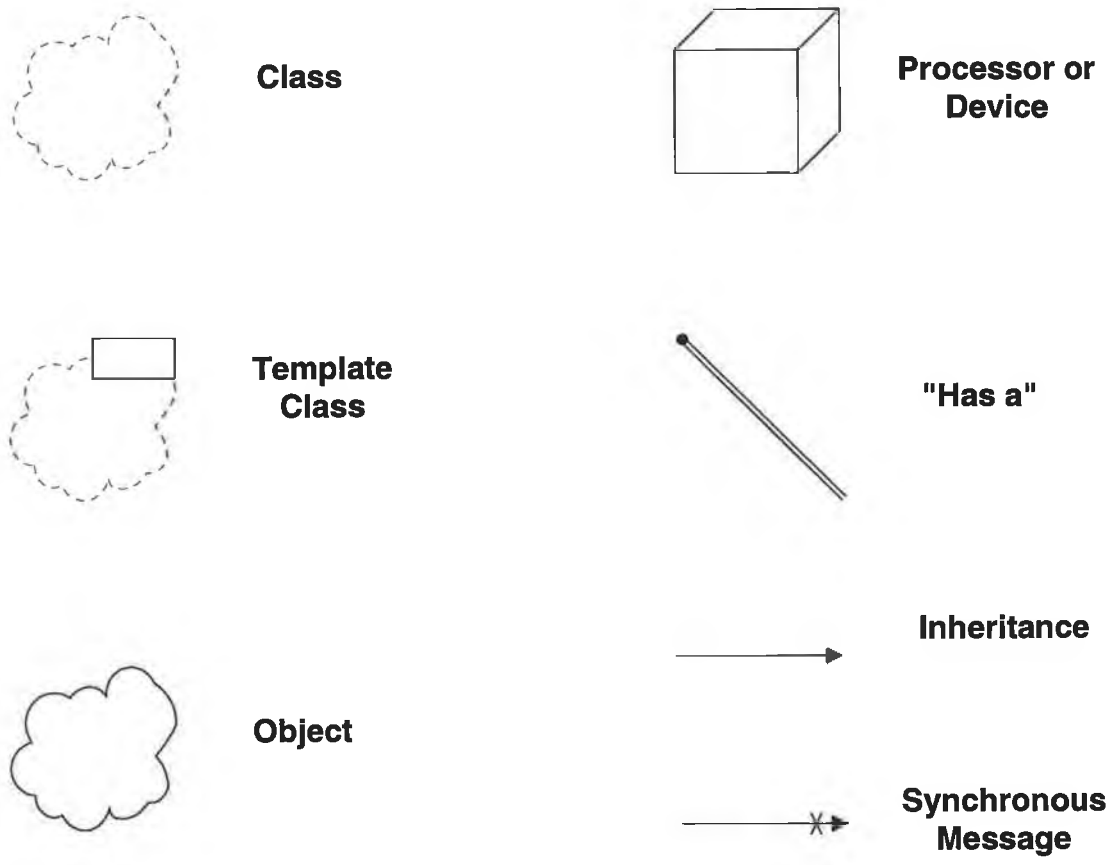


Figure 13. A quick explanation of Booch diagrams and symbols

4.4.2 DistProc Server

4.4.2.1 Architecture

For reasons of portability, the DistProc Server is based on a sequential design. Therefore, the Server can handle only one transaction at a time. This could be a performance bottleneck, but since all operations carried out by a Server in response to a message are short and do not consume much time this is acceptable. This approach avoids the overhead of multiple processes and the additional cost of an interprocess communications mechanism (e.g., shared memory, sockets, etc.) needed for co-ordinate. Basically, the Server is an event (message) driven system with a single input queue; each message is dealt with in sequence (FIFO).

In order to make decisions, such as process distribution etc., the state of the system and all of its components must be maintained. This role is filled by the various resource managers and system tables. Each resource group in the system has an associated system table which is responsible for maintaining the relevant information (e.g., DistProc Slaves, or Remote Processes), while resource managers provide the mechanisms which implement the system's policy decisions.

DISTPROC Slaves and Clients can be dynamically added to and deleted from the virtual machine at any time, excepting the DISTPROC Server, which must be present for the duration.

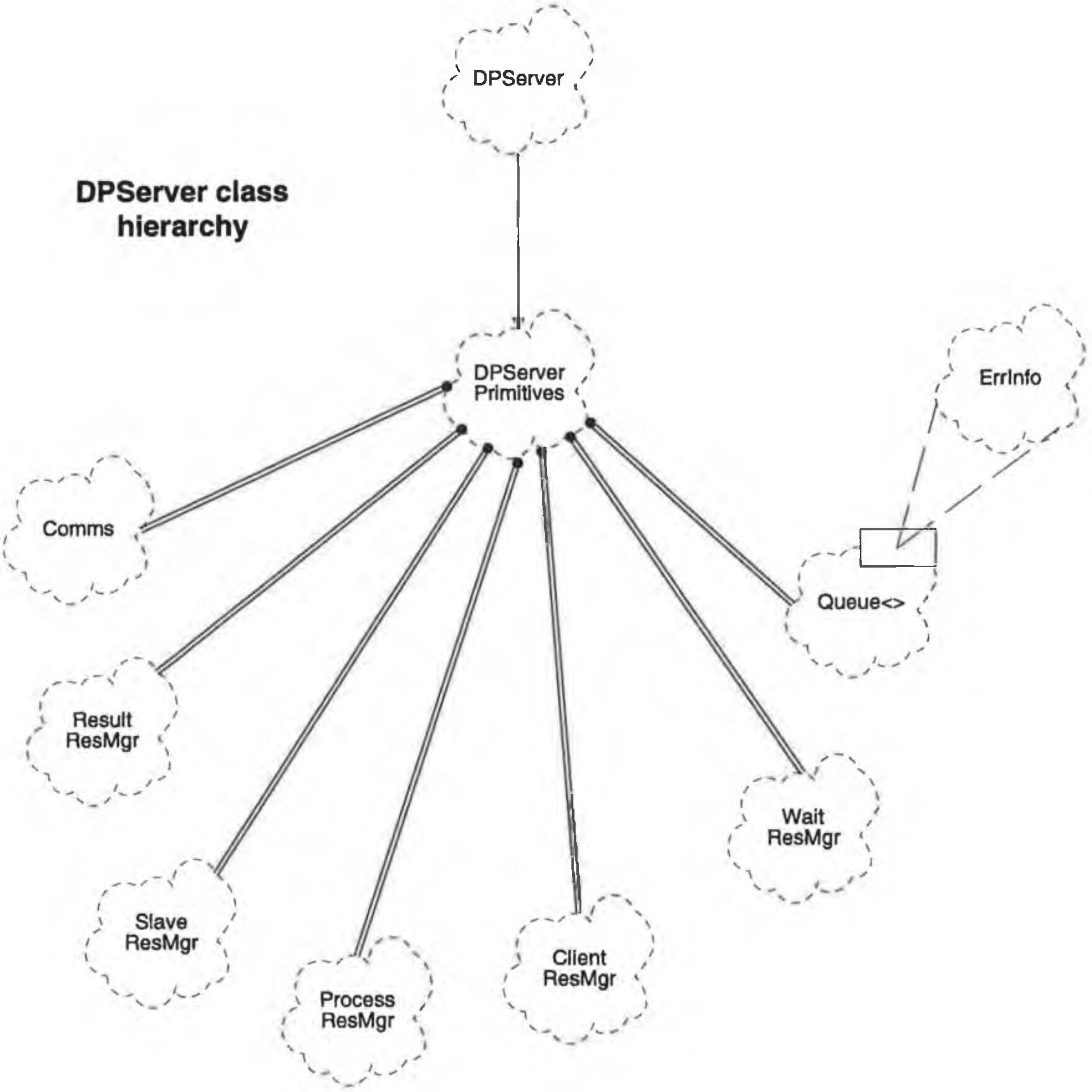


Figure 14. DISTPROC Server Class Diagram.

4.4.2.2 Resource Management

The DISTPROC Server needs to keep a wide range of information about the state of the system in order to make decisions. Resource Managers are used to make system decisions. Each Resource Manager has its own System Table which is used to keep track of the necessary information for each resource. In all there are five Resource Managers - RemProcResMgr, ClientResMgr, SlaveResMgr, ResultResMgr, and WaitResMgr which are used for decisions made regarding Remote Processes, Clients, Slaves, Results, and Wait transactions respectively.

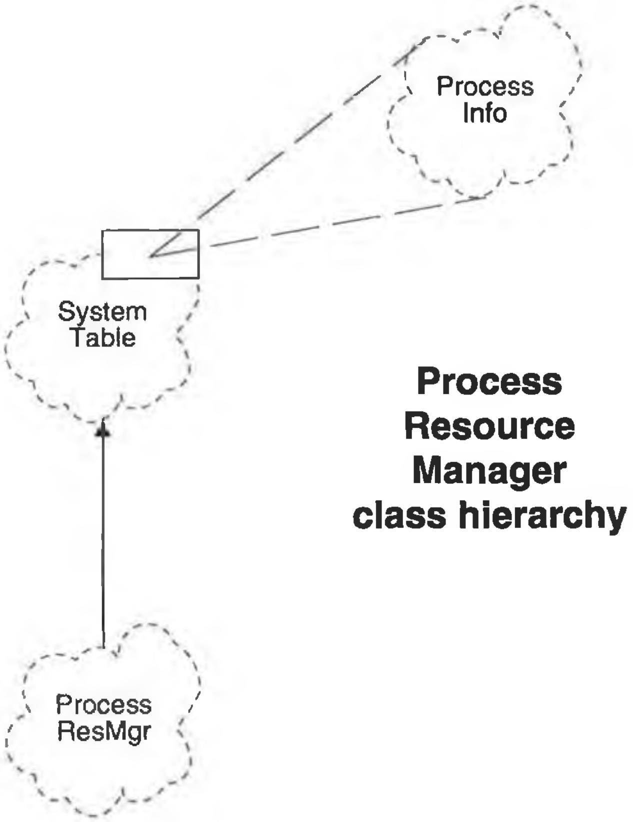


Figure 15. Remote Process Resource Manager Class Diagram

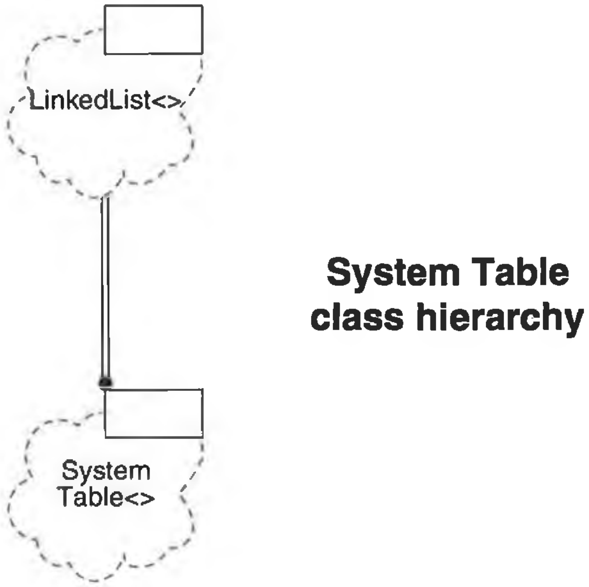


Figure 16. DistProc Server System Table Class Diagram

4.4.2.3 Fault Tolerance

If an error occurs while a DISTPROC Server is dispatching a message (i.e. carrying out the necessary operations) the information on the error is saved, and control is passed to the top level of the Server system, where the error can be more easily dealt with.

The DISTPROC Server daemon keeps information, which is available on request, about aborted Remote Processes.

The role of the DISTPROC Server in fault detection is to convert a Slaves failure, which might ordinarily cause an application to hang forever, into a detectable event, such as a message. No method of handing over the duty of a Server has been implemented. So, if a DISTPROC Slave loses contact with its Server, it cleans up and exits, knowing that the Server will mark it dead.

Failure of a Slave or Client is detected when trying to send it a message. Time-outs for messages sent eventually trigger and the DISTPROC Server gives up. It marks the failed Slave/Client in the appropriate System Tables and tries to complete any pending operations elsewhere, using information stored in the various System Tables. Any further requests for the failed host will return immediately with an error status.

Each DISTPROC Server has an idle timer and periodically sends messages to DISTPROC Slaves so that traffic never goes to zero and Slave host failure detected.

Once a host fails the DISTPROC Slave that was running on it is considered dead forever. Another DISTPROC Slave may be started on the same host at a later time, but it will be considered a different one. There is no way to reclaim any tasks running on the failed host but they can be restarted from scratch.

4.4.3 DISTPROC Slaves

4.4.3.1 Architecture

The role of the DISTPROC Slaves in the system is to execute Remote Processes for a DISTPROC Server on behalf of a DISTPROC Client Application. It is the DISTPROC Slave that is ultimately responsible for carrying out a DISTPROC Client's requests, for example running and killing a Remote Process. The DISTPROC Slave also keeps the DISTPROC Server informed in the case of the abnormal termination of a Remote Process.

4.4.3.2 Process Management

The DISTPROC Slave uses the normal operating system process primitives to carry out the DISTPROC Remote Process primitives. The RunProcessMsg results in calls to fork(), execlp(), and the rsh() functions. The fork() starts a new process, execlp() loads the required Remote Process, and rsh() provides the security. The KillProcessMsg eventually leads to a call to the signal() function on the appropriate DISTPROC Slave; this kills off the required process. The only message that a DISTPROC Slave generates regarding processes, is the DOCMsg; this only happens if a child process is abnormally terminated.

4.4.3.3 Fault Tolerance

The role of the slave with respect to fault tolerance is to monitor child processes and only report to the DISTPROC Server cases of abnormal termination. Such cases are discovered by examining the status fields in the death of child signal which is generated for the death of every child process.

If an error is detected, the DISTPROC Slave sends a DOCMsg to the DISTPROC Server. This message contains the child's process id and the DISTPROC Slaves own SlaveID.

4.4.4 DISTPROC Clients / Applications

4.4.4.1 The Client's System Architecture

Access to the DISTPROC System is only possible through the DISTPROC Client Application Library. The Client Application Library provides a straight-forward interface to the system and is callable from C++.

Operations are made as atomic in nature as possible. For example, when run_process(...) is called it sends a message to a DISTPROC Server requesting it to execute a particular process and waits for an acknowledgement before it continues. No other message will arrive in the meantime.

The Client module is comprised of three main components; the Event Queues (System and Application queue), the Event Manager and the Main process.

4.4.4.2 System Interface

The functions provided by the system allow programmers to connect, disconnect, run processes, kill processes, and wait for processes to finish. Functions are also provided to help with data marshalling. The interface is discussed further in section 4.5.

4.4.4.3 Handling Errors and Tracing Execution

When an error occurs in one of the Client library routines, an error code is returned which indicates the nature of the error. If more debug information is needed then the DISTPROC Client Library can be compiled with the -DDEBUG_ON flag. This is the extent of the debugging support in DISTPROC.

4.4.5 Assumptions about the execution environment

The best environment for the system is a multi-tasking one and it is preferable that the Server and Slaves run on such. The DISTPROC Client Applications can run in a single tasking environment.

The messaging system is built for TCP/IP, but can be changed without too much difficulty.

Currently, the system is running on an Unix network of IBM RS/6000s running AIX 3.2. The whole system can be easily ported to other Unix systems, while the DISTPROC Slaves are portable to most any multi-tasking system, and the Client library can be moved to nearly any platform. The main difficulty in moving platforms is data representation, which changes from processor to processor and operating system to operating system. To overcome this problem some form of platform independent data representation mechanism must be implemented, for example Sun's External Data Representation (XDR).

4.5 DISTPROC Programming Interface

DISTPROC provides primitives for process creation, termination, and synchronisation, as well as data marshalling and system enquiry.

4.5.1 Connecting/Disconnecting

To connect to a DISTPROC Server simply create an instance of the DPClient class. For example:

```
DPClient * dp;
dp = new DPClient(srvhostname);  // create a DPClient object & connect to DPServer
if (dp->Registered() == FAILED)
        exit(-1);
```

'srvrhostname' is the name of the computer running the DISTPROC Server we wish to connect to. This will take care of enrolling the Client program with the specified Server by sending the ClientConnectMsg and setting up the necessary data structures.

Disconnecting from a DISTPROC Server is handled automatically by the DPClient's destructor. When a DPClient object is deleted:

```
delete dp;          // deletes the DPClient object & disconnects from DPServer
```

the destructor sends a ClientDisconnectMsg to the Client's DISTPROC Server, before cleaning up.

4.5.2 Marshalling Services

Message buffers and the Buffer class. A template based approach to saving objects is used. The following data types can be saved:

1. all fundamental data types in C++
2. arrays and memory blocks of any fundamental data type

To handle additional data types/structures simply over-ride the SAVE and LOAD functions.

SAVE template functions are used to save an object to a buffer.

```
// Integral data types - int, char, double, float etc.
template <class T> void SAVE (Buffer &b, T elem)

// Pointer to block of memory - Array or pointer. Won't take 'void *'.
template <class T> void SAVE(Buffer &b, T * elem, size_t num_elems)
```

LOAD template functions are used to reconstruct objects from a buffer.

```
// Integral data types - int, char, double, float etc.
template <class T> Buffer LOAD (const Buffer &b, T &elem)

// Pointer to block of memory - Array or pointer.
template <class T> Buffer LOAD (const Buffer &b, T * elem, size_t num_elems)
```

4.5.3 Process Control

All processes within DISTPROC are represented by an identifier called a DPID, which is a distributed process id number and is the only supported method of identifying processes.

4.5.3.1 Process Creation and Execution - Run

The DPClient::run() member function expects three parameters. The first one, filename, is the name of the file / remote process that is to be executed remotely. The second parameter is a buffer which contains the marshalled input to the new remote process, and the last parameter is the address of a buffer where the result is to be stored.

```
        int       num1, num2, answer, rc;
        Buffer    inbuf, resbuf;
        DPID      rpid;

        SAVE (inbuf, num1);
        SAVE (inbuf, num2);
```

```
if ( ( rpid = dp->run ("remproc1", inbuf, &resbuf) ) == INVALID_DPID)
          cerr << "\nCouldn't start remproc1 running!" << flush;
```

To prepare to send data it must first be packed into a buffer, and is done by calling the appropriate SAVE() functions. Message transmission is handled automatically by the DISTPROC Client Library.

4.5.3.2 Process Synchronisation - Wait

The DPClient::wait() member function expects one parameter, the DPID of the process to wait.

```
if ( (rc = dp->wait(rpid) ) == FAILED)
          cerr << "\nError waiting for remproc1";
else
          resbuf = LOAD(resbuf, answer);
```

After wait() returns successfully the result buffer must be unpacked by using the appropriate LOAD() functions.

4.5.3.3 Process Termination - Kill

The DPClient::kill() member function takes one parameter - the DPID of the process to kill, and returns whether it was successful or not.

```
if ( (rc = dp->kill (rpid) ) == FAILED)
          cerr << "\nCouldn't kill remote process #" << rpid;
```

4.5.4 Querying Services

The DPClient::query() member function is used to obtain information about the state of the system. To obtain information about the DISTPROC system call ::query() with a QueryInfo object as the only parameter.

```
QueryInfo  qinfo;
if ( (rc = dp->query (qinfo) ) == FAILED)
          cerr << "\nCouldn't get system information";
else
          cout << "\nThere are " << qinfo.getNumSlaves() << " available and"
                   "\nthere are " << qinfo.getNumRemProcs() << " running.\n";
```

QueryInfo is used to hold information on the system such as the number of currently available Slaves and the number of Remote Processes executing on the system.

4.6 Observations - User Wariness & Reluctance

Some users may be reluctant to allow other processes to execute on their computer for two main reasons. Firstly, they believe that the background process could damage their system. However, since the background process is executing in its own account and user address space, the risks of it causing damage are the same as another user logging in remotely and subsequently causing damage. Secondly, users worry that it will impinge on system performance, however the system is designed with other users in mind and only uses idle processor cycles for computation. Processes in DISTPROC are executed at a lower priority than normal and cause the minimum of disruption to others.

4.7 Conclusions and Future Research

The DISTPROC System has achieved many of its original design goals. It is a small, easy to run system, portable and could possiby be extended to support a heterogeneous environment. There is limited support for fault tolerance, scalability, and performance.

Looking beyond the original goals however there are many new features that could be added. Currently there is no support for execution tracing, checkpointing, dynamic load balancing or advanced parallelism primitives. To develop these features using the current approach of building a distributed system around an existing language such as C/C++ is difficult, both from the developers' and end users' point of view.

Fortunately, the area of distributed computing is changing rapidly with the move to distributed interoperable object technology. Distributed objects provide a means of overcoming many of the difficulties involved in distributing computing systems. However, there would still be a need for a system, such as the DISTPROC System, to control the distributed objects and to provide process management (distributed scheduling, etc.).

In conclusion, DISTPROC is useful for investigating distributed computing and for connecting a small number of workstations together for applications with well defined, independent and computationally expensive components and low communications requirements.

5. Cryptography

5.1 Introduction to Cryptography

Cryptography was once of interest primarily to the military and diplomatic services. Today it is particularly important in the area of computing [Die90]. Eavesdropping is becoming easier as huge volumes of business are handled over communications networks and the use of electronic mail and electronic funds transfer (EFT) is increasing.

Cryptography is the use of data transformations to make the data incomprehensible to all except its intended users. The privacy problem is concerned with preventing the unauthorised extraction of information from a communication channel. The authentication problem is concerned with preventing an adversary from modifying a transmission or inserting false data into a transmission. The problem of dispute is concerned with providing the receiver of a message with legal proof of the sender's identity, i.e. the electronic equivalent of a written signature.

Two of the most popular encryption schemes in use today are the Data Encryption Standard (DES) [NBS77] and the Rivest, Shamir, and Adleman (RSA) scheme [Riv78]. DES is a symmetric encryption scheme - the same key is used for encryption and decryption, and RSA is an asymmetric encryption scheme - different keys are used for encryption and decryption.

Cryptanalysis is the process of attempting to regenerate plaintext from ciphertext but without knowledge of the decryption key; this is the normal task of the eavesdropper. If the eavesdropper, or cryptanalyst, cannot determine the plaintext from the ciphertext (without the key), then the cryptographic system is secure.

5.1.1 What is encryption?

Encryption is a process used to scramble a message, so that only the sender and legitimate recipient knows what it contains [Lia93]. Only the recipient knows the secret method for decrypting the message, so that when it is received it can be restored to its original form. Its purpose is to ensure privacy by keeping information hidden from anyone for whom it is not intended, even those who can see the encrypted data.

Encryption allows secure communication in multi-user environments. Suppose two people, A and B, want to send messages to each other without anyone else being able to read it. A encrypts the message, called the plaintext, with an encryption key and sends the encrypted message, called the ciphertext, to B. To read the plaintext B must decrypt the ciphertext using the decryption key. An attacker can try to obtain the secret key or to recover the plaintext without using the secret key. If the cryptosystem is secure then the attacker cannot recover the plaintext from the ciphertext except by using the decryption key.

5.1.2 Public key cryptography

Throughout history many complex systems have been used to keep information secret from prying eyes. These systems have all been based on some form of cryptographic algorithm and a secret key. If you have the key then you can encrypt and decrypt messages. This method is known as secret-key cryptography. The weakness of this system is getting the sender and receiver to agree on the secret key without anyone else finding out. Anyone who overhears or intercepts the key in transit can later read all messages encrypted with the key. The generation, transmission and storage of keys are called key management. Secret-key cryptosystems often have difficulty providing secure key management.

Diffie and Hellman [Dif76] changed all this when they introduced Public-Key Cryptography. Public key cryptography solves the problem of key management by using two keys, one public and the other private, instead of the more traditional one key algorithm. Each person's public key is published while the private key is kept secret. The necessity for sender and receiver to share secret information is eliminated as is the need to trust some form of communication. Any person with the public key can encrypt messages, but it can only be decrypted with the private key that is in the sole possession of the intended receiver. It is computationally impossible to deduce the private key from the public key.

The benefits of public-key cryptography over secret-key cryptography include increased security, authentication and privacy. Security is increased as the private keys need never be transmitted or revealed to anyone. In secret-key systems there is always the possibility that the secret key may be divulged in transit. Authentication can be provided by way of digital signatures. Authentication by secret-key systems requires the sharing of some secret and sometimes requires the trust of a third party as well whereas in public-key systems each user has sole responsibility for protecting his or her private key. Digitally signed messages can be proved to be authentic to a third party. Using public key cryptography for encryption is slow in comparison to many popular secret-key systems. This disadvantage can be overcome by combining the security advantages of public-key systems and the speed advantages of secret-key systems. The public-key system is first used to encrypt a secret key, it is then used to encrypt a message before sending it to the intended receiver.

The first use of public-key techniques was for secure key exchange in an otherwise secret-key system [Dif76] and this is still one of its main functions.

5.2 Some Maths Background

In order to understand cryptography a basic knowledge of number theory is helpful, so I am just going to give a quick introduction to some of the main points; for a detailed study see [Hua82].

5.2.1 Modular arithmetic

In modular arithmetic all calculations are performed *modulo n,* which basically means the results of all operations are 'reduced modulo *n*', or, reduced to their remainder when divided by *n*.

Modular arithmetic is like generalised clock arithmetic, where if you advance the clock five hours from 10.00, it does not become 15.00 but 3.00. Clock arithmetic is arithmetic modulo 12 (or modulo 60 with the minutes hand), but it is possible to do arithmetic modulo anything. For example, 26 + 9 modulo 27 is 8; 5 * 5 modulo 18 is 7; and 9 * 3 modulo 7 is 6.

Basically, $a \equiv b\ (mod\ n)$ if $a = b + kn$ for some integer *k*. If *a* and *b* are positive and *a* is less the *n*, *a* can be considered the remainder of *b* when divided by *n*. In general, a and b leave the same remainder when divided by *n*. Sometimes, *b* is called the residue of *a*, modulo *n*, and sometimes *a* is called congruent to *b*, module *n* ($\equiv$ denotes congruence); both mean the same thing.

So, two integers *a* and *b* are said to be **congruent modulo** a third integer, *n*, if and only if (iff) their difference is exactly divisible by *n*; alternatively iff they each leave the same remainder on division by *n*. We write $a \equiv b(n)$, for example, $80 \equiv 25(11)$ because 80 - 25 = 55 = 5 * 11.

A negative answer is made positive by adding *n* to it, so the result of any operation is a positive number between 0 and *n*-1. Raising a number to a large power modulo *n* is just a series of multiplications and divisions, as intermediate results can always be reduced.

For example, to calculate a^8 mod *n*, perform three small multiplications and modular reductions, $((a^2 \text{ mod } n)^2 \text{ mod } n)^2 \text{ mod } n$, instead of three multiplications and one big modular reduction, $((a^2)^2)^2 \text{ mod } n$.

Cryptography uses computation modulo *n* a lot, because problems like calculating discrete logarithms and square roots are hard. It is also easier to work with, because it restricts the range of all intermediate values and the result, so we can perform exponentiation in modular arithmetic, without generating huge intermediate results.

The opposite of exponentiation modulo *n* is calculating a discrete logarithm.

5.2.2 Prime numbers

A **prime number**, *p*, is an integer (whole number) which is only exactly divisible by itself and unity (one), for example, 2, 3, 5, ...31, ...613... A integer is said to be **composite** if it is not prime.

There are an infinite number of prime numbers. Large primes are used throughout various cryptography techniques.

5.2.3 Greatest common divisor

Two integers *a* and *b* are said to be **coprime** or **relatively prime** iff they have no common divisor other than 1. We write gcd(a, b) = 1 or simply (a, b) = 1. In general (a, b) denotes the greatest common divisor of *a* and *b*. 16 and 57 are coprime while 96 and 172 are not. A prime number is relatively prime to all other numbers except its multiples. GCDs are usually found using Euclid's algorithm.

5.2.4 Inverses Modulo *n*

Normally it's easy to find the inverse of a number; the multiplicative inverse of 4 is 1/4, as 4 * 1/4 = 1. Finding the inverse of a number modulo *n* is more complicated. For instance, $4 * x \equiv 1 \bmod 7$, is like solving $4x = 7k + 1$, where *x* and *k* are integers.

The general problem is finding an *x* such that $1 = (a * x) \bmod n$, which is also written as $a^{-1} \equiv x \pmod{n}$. For example, the inverse of 5, modulo 14, is 3; $5 * 3 = 15 \equiv 1 \pmod{14}$. However, 2 has no inverse modulo 14.

In general, $a^{-1} \equiv x \pmod{n}$ has a unique solution if *a* and *n* are relatively prime. If *a* and *n* are not relatively prime, then $a^{-1} \equiv x \pmod{n}$ has no solution. If *n* is a prime number, then every number from 1 to n-1 is relatively prime to *n* and has exactly one inverse in that range.

Euclid's algorithm, sometimes called the extended Euclid algorithm, can be used to find the inverse of a number modulo *n*. See [Knu81] for a detailed explanation. Another method for calculating the inverse modulo *n* is via Euler's Totient Function, but it cannot always be used.

5.2.5 Fermat's little theorem

If *m* is a prime, and *a* is not a multiple of *m*, then Fermat's little theorem says that $a^{m-1} \equiv 1$ (mod m).

5.2.6 Euler's Totient Function

Euler's Totient Function (also known as Euler's phi function), Ø(*a*) is defined for all positive integers, *a*, as the number of integers in 0, 1, 2,...*a*-1 which are coprime with *a*. That is the number of positive integers less than *a* and relatively prime to *a* (i.e. not sharing a common factor other than 1). For example, Ø (15) = 8, the eight numbers being 1, 2, 4, 7, 8, 11, 13 and 14, and Ø(4) = 2; Ø(*a*) = *a* -1 when *a* is prime, Ø(5) = 4.

If *a* and *m* are positive integers such that (*a*, *m*) = 1 then the smallest positive integer *k* such that $a^k \equiv 1(m)$ (i.e. a^k leaves the remainder 1 when divided by *m*) is called either the order of *a* modulo *m* and written $\text{ord}_m a$, or the exponent to which *a* belongs modulo *m*. For example, 7 belongs to exponent 2 modulo 4, i.e. $7^2 \equiv 1(4)$.

If, however, *k* = Ø(*m*) then *a* is called a **primitive root** modulo *m*. For example, 3 is a primitive root to moduli 17, 289, and 578.

5.2.7 Quadratic residues

If *p* is prime and *a* is less that *p*, (i.e., *a* and *p* are coprime) then *a* is called a **quadratic residue** modulo *p* if $x^2 \equiv a \pmod{p}$, for some *x*. For example, 13 is a quadratic residue modulo 23 because $6^2 \equiv 13(23)$. However, not all values of *a* satisfy this property. For example, if *p* = 7, the quadratic residues are 1, 2, and 4:

$1^2 = 1 \qquad \equiv 1 \pmod 7$

$2^2 = 4 \qquad \equiv 4 \pmod 7$

$3^2 = 9 \qquad \equiv 2 \pmod 7$

$4^2 = 16 \qquad \equiv 2 \pmod 7$

$5^2 = 25 \qquad \equiv 4 \pmod 7$

$6^2 = 36 \qquad \equiv 1 \pmod 7$

Each quadratic residue appears twice on this list. There are no values of *x* that satisfy any of these equations:

$x^2 = 3 \pmod 7$

$x^2 = 5 \pmod 7$

$x^2 = 6 \pmod 7$

The quadratic nonresidues modulo 7 are 3, 5, and 6.

5.3 Introduction to RSA

Of all the public-key algorithms proposed over the years, RSA is by far the easiest to understand and implement and the most popular. Named after its inventors, Ron Rivest, Adi Shamir, and Leonard Adleman, who first introduced the algorithm in 1978, it has withstood intensive scrutiny; although the analysis neither proved nor disproved security it does indicate a high level of confidence in the algorithm.

The method for encrypting messages is public knowledge. Anyone who wants to receive a message gives out a pair of specially prepared numbers which can be used in a standard fashion to encrypt all the messages sent to the receiver. The numbers are chosen in conjunction with another one that the receiver keeps private and uses to decrypt the message. No-one can reverse the process without the private number.

5.3.1 Primes and RSA

RSA gets its security from the difficulty of factoring large numbers. The public and private keys are functions of a pair of very large (100 or more digits) prime numbers. The algorithm calculates both keys from the prime numbers, and determining one key from the other is conjectured to be equivalent to factoring the product of the two primes.

5.3.2 How RSA Works

RSA works as follows [Sch92][Fah94]: to generate two keys choose two large prime numbers p and q and find their product $n = pq$ which is called the modulus. Then randomly choose the public key, e, less than n and relatively prime to $(p\text{-}1)(q\text{-}1)$ (i.e. e has no factors in common with $(p\text{-}1)(q\text{-}1)$). The easiest way to do this is to select another prime number for e that is larger than both $(p\text{-}1)$ and $(q\text{-}1)$. Finally, compute the private key, d, such that $e.d = 1$ mod $((p\text{-}1).(q\text{-}1))$. In other words, $d = e^{-1}$ mod ((p-1).(q-1)). The numbers e and d are called the public and private exponents, respectively. The public key is the pair (e, n) and the pair (d, n) is the private key.

5.3.3 How To Use RSA

To perform encryption, the sender must first divide the message m into numerical blocks, by using the characters ASCII codes for example, such that each block has a unique representation modulo n (with binary data, choose the lowest power of 2 less than n). That is, if both p and q are hundred digit primes, then n will have about 200 digits, and each message block, m, should be 200 digits long. The encrypted message, c, will be made up of similarly sized message blocks c_i of about the same length. The encryption formula is simply c_i =

m_i^e(mod *n*). Once all blocks have been processed the resulting ciphertext can be safely sent to the recipient who can then decrypt it using the secret key pair (*d*, *n*).

To decrypt the message take each encrypted block c_i and compute $m_i = c_i^d$ (mod *n*). With all blocks of digits decrypted, the message can be put back together by reversing the original pre-encryption organisation, typically by interpreting the numbers as ASCII codes and converting back to characters.

The message could just as easily have been encrypted with *d* and decrypted with *e*.

5.3.4 The Security of RSA

The security of the system is totally dependent on the fact (as yet unproved) that there exists no fast algorithm (not even a probabilistic one, as in generating primes) for factoring large numbers.

Any adversary will have the public key *e* and the modulo *n*. To find the decryption key, *d*, *n* must be factored. At the moment the best factoring algorithms take on the order of O($e^{\text{sqrt}(\ln n * \ln(\ln n))}$) steps to solve. If *n* is a 200 bit number, factoring will take on the order of 2.7 * 10^{11} steps; for a 664 bit *n* (200 digits) about 1.2 * 10^{23} steps. Assuming a computer can perform a million steps per second it will take 3.8*10^9 years to factor a 200 digit number. If you need more security, increase the length of *n*.

This is an area of great interest to computer scientists and mathematicians, and although it all remains unproved, the general consensus is that these types of encryption systems are safe.

5.3.5 Breaking RSA

There are a few possible interpretations of what it take to break RSA. The most serious would be for an attacker to discover the private key corresponding to a given public key as this would allow complete access. The most obvious means of attack is to factor *n*, the public modulus, into its two prime factors, *p* and *q*. The private key, *d*, can then be easily found from *p*, *q*, and *e*, the public exponent. The hard part is factoring *n*, and this is what the security of RSA depends on. The task of recovering the private key is equivalent to the task of factoring the modulus; you can use *d* to factor *n*, as well as use the factorisation of *n* to find *d*.

Another way to break RSA is to find a technique to compute e-th roots mod *n*. Since $c = m^e$, the e-th root of *c*, the ciphertext, is the message *m*. This attack would allow encrypted message to be recovered and signatures forged without knowing the private key. This attack is not known to be equivalent to factoring and currently no such methods are used against RSA.

Aside from the brute force method of trying every key the above attacks are the only ways to break RSA so that all messages encrypted under a given key are recoverable. Other methods aim to recover single messages in isolation; other messages encrypted under the same key would remain locked.

The simplest single message attack is the guessed plaintext attack. An attacker sees a ciphertext, guesses what the message might be and encrypts this guess with the public key of the recipient. By comparing the actual ciphertext with the guessed ciphertext the attacker will know if the guess was correct. This attack can be thwarted by appending some random bits to the message. Another single message attack discussed by [Has88] can occur if someone sends the same message to three others, who each have public exponent $e = 3$. An attacker who knows this and sees the three messages will be able to recover the message m using the Chinese Remainder Theorem. There are also some "chosen ciphertext" attacks in which the attacker creates some ciphertext and gets to see the corresponding plaintext as explained by [Dav82].

There are also attacks that aim not at RSA itself but at insecure key management. For example, if someone stores his private key insecurely, an attacker may discover it.

5.3.6 Strong Primes

It has often been suggested that in choosing a key pair, one should use "strong" primes p and q to generate the modulus n. Strong primes are those with certain properties that make the product n hard to factor by specific factoring methods; such properties have included, for example, the existence of a large prime factor of p-1 and a large prime factor of p+1. The reason for these concerns is that some factoring methods are especially suited to primes p such that p-1 or p+1 has only small factors; strong primes are resistant to these attacks.

However, recent advances in factoring appear to have obviated the advantage of strong primes; the elliptic curve factoring algorithm is one such advance. The new factoring methods have as good a chance of success on strong primes as on "weak" primes; therefore, choosing strong primes does not significantly increase resistance to attacks. However, new factoring techniques may be developed in the future which are based on certain properties; if so, then choosing strong primes may help to increase security.

5.3.7 Key Size

The two primes p and q, which compose the modulus, should be roughly equal in length. This will make the modulus harder to factor than if one of the primes was very small. Thus if one chooses to use a 512-bit modulus, the primes should each have a length approximately 256 bits.

5.3.8 Alternatives to RSA

A look at the annual Crypto and Eurocrypt proceedings shows that many other public key cryptosystems have been proposed. A mathematical problem called the knapsack problem was the basis for several systems, but these have lost favour as many versions have been broken. ElGamal is another popular system that is based on the discrete logarithm problem.

5.3.9 Protocols and Applications of Public Key Cryptography

Public key cryptography is currently used in a plethora of areas which have implications far beyond simple data encryption [Sch92]. It allows people to do things securely over computer networks that are impossible in other ways such as password protection, digital signatures, simultaneous contract signing and oblivious transfer.

5.3.9.1 Password Protection

Conventional password protection schemes, where the host computer stores the password in encrypted form, have serious security problems. For one, when the user types his password into the system, anyone who has access to his data path can read it. Secondly, anyone with access to the processor memory can see the password before the system encrypts and compares it with the encrypted password in the password file.

Public-key cryptography solves the problem by allowing the host computer to keep a file containing every user's public key; each user keeps his own private key. When logging in the host sends the user some random strings, which the user encrypts with the private key and returns to the host. The host then decrypts the message using the user's public key. If the decrypted message matches the original message then authentication is complete.

5.3.9.2 Digital Signatures

With traditional paper communications the problems of authentication are all handled by formal written signatures on documents, but for electronic communications an electronic signature is needed.

A property of public-key cryptography is that either key can be used for encryption. Encrypt a document with your private key, and you have a secure digital signature. Anyone with your public key can decrypt the document, so anyone can read it. Only you have access to your private key so no one else could have signed it. Also, no one can modify the encrypted document as any modification to the encrypted document will produce gibberish when decrypted.

The problem with this protocol is that generating a public-key cryptography digital signature on an entire document takes a lot of time. It is easier to use a one-way hash function on the document, which produces a small fingerprint, and then sign the fingerprint with a private key.

5.3.9.3 Improved Key Exchange

Using digital signatures during a DES-key exchange protocol circumvents a potential security breach. Consider the situation where an adversary intercepts data from Alice and Bob. The adversary could pretend to be Alice and send a Bob a different DES key. Since Bob's public key is public he could be easily misled, complete the protocol, and use the new key to encrypt his messages to "Alice". The adversary would then be able to read all the data Bob sends to Alice. The adversary could also fool Alice using the same technique. If the adversary is quick enough he can decrypt Bob's data and reencrypt it for Alice, and then decrypt Alice's data and reencrypt it for Bob. Neither Alice nor Bob would have any idea that their supposedly secure communications were being intercepted by a third party. This is called "man-in-the-middle" attack.

With digital signatures, a central trusted authority can sign both Alice's and Bob's public keys. The signed keys would include a signed certification of who they belonged to. Now both know that the public key they received over the communications link actually belongs to the other person. The DES key exchange can then proceed. Finally, to ensure that neither Alice nor Bob are not impostors, both should initiate the challenge and reply protocol described in the password protection example.

5.4 References

[Dav82] Davida, G., "Chosen signature cryptanalysis of the RSA public key cryptosystem." Technical Report TR-CS-82-2, Dept of EECS, University of Wisconsin, Milwaukee, 1982.

[Die90] Dietel, H.M., "Operating Systems", 2nd Ed., Addison-Wesley, 1990.

[Dif76] Diffie, W., and M. Hellman. "New Directions in Cryptography", IEEE Transactions on Information Theory, IT-22:644-654, November 1976.

[Fah94] Fahn, Paul, "FAQ about today's Cryptography," pp. 5. Available from: http://www.rsa.com/rsalabs/faq/StartHere.html

[Has88] Hastad, J., "Solving simultaneous modular equations of low degree." SIAM J. Computing, 17:336-341, 1988.

[Hua82] Hua, L.K., "Introduction to Number Theory," Berlin: Springer-Verlag, 1982.

[Kal93] Kaliski, B. "A Survey of Encryption Standards", IEEE Micro (December 1993).

[Knu81] Knuth, D.E., "The Art of Computer Programming: Vol. 2, Semi-numerical Algorithms," 2nd Ed., Addison-Wesley, 1981.

[Lia93] Liardet, M., "Low Level:- Cryptic Clues", Personal Computer World, Nov. 1993.

[NBS77] National Bureau of Standards, "Data Encryption Standard," January 1977, NTIS NBS-FIPS PUB 46.

[Rib91] Ribenbiom, Paulo "The Little Book of Big Primes," Springer-Verlag, 1991, ISBN 0-3-540-97508-X.

[Riv78] Rivest, R.; A. Shamir; and L. Adleman, "On Digital Signatures and Public Key Cryptosystems," Communication of the ACM, Vol. 21, No. 2, February 1978, pp. 120-126.

[Sal90] Salomaa, A. "Public-key Cryptography", Berlin Germany: Springer-Verlag, 1990.

[Sch92] Schneier, B. "Untangling Public-key Cryptography", Dr. Dobb's Journal (May 1992).

[Sch94] Schneier, Bruce, "Applied Cryptography - Protocols, Algorithms, and Source Code in C," Wiley, 1994.

[Sco89] Scott, Michael, "Factoring Large Integers on Small Computers," Working Paper: CA-0169, School of Computer Applications, Dublin City University, Dublin, Ireland.

6. Factoring

6.1 Introduction

Factoring is the act of splitting an integer into a set of smaller integers (factors) which, when multiplied together, form the original integer. For example, the factors of 15 are 3 and 5; the factoring problem is to find 3 and 5 when given 15. Prime factorisation requires splitting an integer into factors that are prime numbers; every integer has a unique prime factorisation. Multiplying two prime integers together is easy, but as far as is known, factoring the product is much more difficult.

Factoring is the underlying, presumably hard problem upon which several public-key cryptosystems are based, including RSA. Factoring an RSA modulus would allow an attacker to figure out the private key and thereby allowing anyone who can factor the modulus to decrypt messages and forge signatures. The security of RSA therefore depends on the factoring problem being difficult. Unfortunately, it has not been proven that factoring must be difficult, and there remains a possibility that a quick and easy factoring method might be discovered, although factoring researchers consider this possibility remote.

Factoring large numbers takes more time than factoring smaller numbers. This is why the size of the modulus in RSA determines how secure an actual use of RSA is; the larger the modulus, the longer it would take an attacker to factor, and thus the more resistant to attack the RSA implementation is.

Around 1978 Rivest et al. discovered that the difficulty of breaking certain cryptographic codes depends on the difficulty of factoring large numbers. This discovery renewed interest in the classical problem of the factorisation of integers. In 1974 it was considered very difficult to factor numbers in the 40-50 decimal digit range. Now, 20 years later, the factorisation of 100 digit numbers and larger is possible. This shows the huge progress over the intervening years, particularly when we take into account the - experimental - fact that if the number of decimal digits of the number to be factorised is increased by three, then the amount of CPU time needed is roughly doubled.

6.2 The Current State of Factoring

In the last decade factoring has become easier for two main reasons: computer hardware has increased in power, and newer, better factoring algorithms have been developed.

Improvements in hardware will continue unabated and will allow ever larger numbers to be factored. When considering public-key cryptosystems this point is largely redundant; simply increasing the size of the public and private keys will compensate for any hardware improvements. This rule may fail in the case where fast machines of the future are used to factor old keys; in which case the attacker gains the advantage. To allay this possibility one should continually update the size of the keys in use and consider using a larger key than is considered prudent.

Better factoring algorithms have been more helpful to attackers of public key cryptosystems than have hardware improvements. As the importance of cryptography has burgeoned, so has interest in the factoring problem, and many researchers have found new factoring methods or improved existing ones. The most successful factoring attempts in recent years have used a wide range of distributed systems from computer clusters to the internet [Len90a] to divide the workload among multiple computers and thereby reduce the time taken to factor a number. This has made factoring easier for numbers of any size, however factoring remains a very difficult problem.

Overall, any recent decrease in security due to algorithm improvement can be offset by increasing the key size. As long as hardware continues to increase in performance at a faster rate than that at which the complexity of factoring algorithms decreases, the security of public key cryptosystems will increase or at the minimum result in no net loss in security, assuming key sizes are increased accordingly. The open question is how much faster factoring algorithms can get; there must be some intrinsic limit to factoring speed, but the limit remains unknown.

6.3 Factoring Methods

Factoring [Fah94] [Sch94] [Sco89] is a very active field of research among mathematicians and computer scientists. Factoring algorithms measured by their big-O asymptotic efficiency. O notation measures how fast an algorithm is; it gives an upper bound on the number of operations (to order of magnitude) in terms of *n*, the number to be factored, and, *p*, a prime factor of *n*. See [Cor90] for an introduction to big-O notation and [Bre89] for a discussion of factorisation.

A "general number" is one with no special form that might make it easier to factor; an RSA modulus is a general number. Note that a 512-bit number has about 155 digits. Numbers which are composed of two prime divisors of approximately equal size are the hardest to factorise, and are particularly interesting for cryptography.

Factoring algorithms come in two flavours, special and general purpose; the efficiency of the former depends on the unknown factors, whereas the latter depends on the numbers to be factored. Special purpose algorithms are best for factoring numbers with small factors, but the numbers used in public-key cryptosystems (e.g., RSA) should not have any small factors. Therefore, general purpose factoring algorithms are the more important ones in the context of cryptographic systems and their security.

Special purpose factoring algorithms include the Pollard rho method [Pol75], with expected running time of O(sqrt(*p*)), and Pollard's *p*-1 method [Pol74], with expected running time O(*p*'), where *p*' is the largest prime factor of *p*-1. Both of these take an amount of time that is exponential in the size of *p*, the prime factor that they find; thus these algorithms are too slow for most factoring jobs. Pollard's (*p* - 1) method specialises in quickly finding a factor *p* of a number *N* for which (*p* - 1) has itself only small factors. William's [Wil82][Mon87] factoring algorithm is similar to Pollard's (*p* - 1) method, but can find a factor *p* of *N* for which (*p* + 1) has only small factors. Another special purpose factoring algorithm is Brent-Pollard [Bre80] which is good for finding factors in the range 10^4 - 10^10.

The elliptic curve method (ECM) [Len87] is superior to these; its asymptotic running time is $O(\exp(\sqrt{2 \ln p \ln \ln p}))$. The ECM is often used in practice to find factors of randomly generated numbers; it is not strong enough to factor a large public-key modulus. ECM's computing time depends on the size of the smallest prime factor of the number to be factorised.

The multiple polynomial quadratic sieve (mpqs) [Sil87] algorithm, the fastest general purpose factoring algorithm, is capable of factoring numbers between 110 and 135 digits irrespective of the size of the factors and has a running time in the order of $O(\exp(\sqrt{\ln n \ln \ln n}))$. The mpqs technique (and some of its variants) has successfully factored numbers greater than 110 digits; a variation known as ppmpqs [Len91] has been particularly popular.

The best special purpose factoring algorithm is the number field sieve [Len90b][Buh92], which runs in approximately $O(\exp(1.9(\ln n)^{1/3}(\ln \ln n)^{2/3}))$. This is the fastest known factoring algorithm and has recently been implemented by [BLZ94]. The Number-Field Sieve works particularly well for numbers of the form *bn+c* with *b* and *c* quite small. The number field sieve is overtaking the mpqs as the most widely used factoring algorithm, as the size of numbers which can be factored increases from about 120 digits to 130 or 140 digits.

Numbers that have a special form can already be factored up to 155 digits or more [Len93]. The Cunningham Project [Bri88] keeps track of the factorisations of numbers with these special forms and maintains a "10 Most Wanted" list of desired factorisations. Another good way to monitor current factoring capability is to look at recent results of the RSA Factoring Challenge.

6.3.1 Distributed Factoring Methods

In recent years a lot of work has gone into developing factoring algorithms that can be easily worked on in parallel. The ECM, MPQS, and NFS algorithms have all been used in a parallel/distributed form by [Dix91], [Len90a], [Len90b] and [Len93].

One of the most impressive efforts to date has been by Lenstra and Manasse [Len93]. Using the Number-Field Sieve, they completed the factorisation of the 155-digit number $2^{512} + 1$, the ninth Fermat number, which turns out to be the product of primes that are 7, 49, and 99 digits in length. The calculations were done on approximately 700 workstations and in one of the final stages a supercomputer was used. The entire factorisation was accomplished in four months. The work involved in factoring the number was distributed via electronic mail, the results returned to a central site. Once all the results were collected the final 'matrix step' of the NFS algorithm was carried out on a 65536-processor Connection Machine.

6.4 Lenstra's Elliptic Curve Method

Lenstra's factorisation method is similar to Pollard's and William's methods, but is potentially much more powerful. It works by randomly generating an Elliptic Curve, which can then be used to find a factor p of N, for which $p + 1 - \delta$ has only small factors, where δ depends on the particular curve chosen. If one curve fails then another can be tried, an option not possible with the Pollard / Williams methods. This method also uses two phases and although it has very good asymptotic behaviour, it is much slower than the Pollard / Williams methods for each iteration.

The elliptic curve method (ecm) [Len90a] consists of a number of independent factorisation trials (curves). Any curve might find a factorisation, independently of any other curve. ECM's probability of success is related to the size of the smallest factor of the number to be factored; the larger the smallest factor is the smaller the probability of success. The elliptic curve method is a special purpose factoring algorithm in that it can only be expected to work if the number to be factored has a reasonably small factor.

The elliptic curve method is a very useful method to find small factors of large numbers, but one doesn't know if a number has a small factor until the number is factored! If ecm is applied to a number that has a small factor, ecm has a reasonable chance of succeeding; if there is no small factor then ecm has little chance of success. This leads to the question of how much time should be spent on a factorisation attempt with ecm.

The time invested in a failed ecm factorisation attempt is completely wasted. A failed result from ecm contributes nothing useful that might help in further factorisation attempts. There is even the possibility if ecm fails, that a small factor does exist, though this is less likely.

So why bother with ecm at all? Firstly, ECM is easily distributed and is suitable for implementing in a distributed parallel environment. It is also useful for doing a preliminary check on big numbers for small factors before using other, more time consuming methods.

6.4.1 Description of the Elliptic Curve Method

The Elliptic Curve Method can be broken down into three stages [Len90a].

Stage 1: Given a number n to be factored, make sure that it's not a prime number, randomly select an elliptic curve modulo n and a point x in the group of points of this elliptic curve.

Stage 2: Select an integer m_1, and raise x to the power k, where k is the product of all prime powers $\leq m_1$. If this computation fails because a non-trivial factor has been found then quit. Otherwise, start all over again on a new curve.

Stage 3: Select an integer $m_2 > m_1$, and try to compute x^{kq} for the primes q between m_1 and m_2 in succession. If this computation fails because a non-trivial factor has been found, then terminate. Otherwise, start all over again.

The limits are calculated from the same formula as used in Yoichi Koyama's [Koy91] Ubasic ECM system.

Every iteration or trial of ECM is completely independent of every other trial and any number of them can be carried out in simultaneously. Although each iteration is quite time-consuming, the asymptotic behaviour of this algorithm is very good. Therefore an algorithm, such as Brent-Pollard [Bre80], that is good at finding small factors should be used to find any small factors before applying the Elliptic Curve method.

6.5 The Future of Factoring

Factoring is strongly believed to be a difficult mathematical problem, although it has not been proved so. Therefore there remains a possibility that an easy factoring algorithm will be discovered. This development, which could seriously weaken RSA, would be highly surprising and the possibility is considered extremely remote by the researchers most actively engaged in factoring research.

Another possibility is that someone will prove that factoring is difficult. This negative breakthrough is probably more likely than the positive breakthrough discussed above, but would also be unexpected at the current state of theoretical factoring research. This development would guarantee the security of RSA beyond a certain key size.

6.6 References

[Bre80] R. Brent, "An improved Monte Carlo Factorisation Algorithm," BIT, 20:176-184, 1980.

[Bre86] R. Brent, "Some Integer Factorisation Algorithms using Elliptic Curves," Australian Computer Science Communications v. 8, 1986, pp 149-163.

[Bre89] R. Brent, "Parallel Algorithms for Integer Factorisation," Research Report CMA-R49-89, Computer Science Laboratory, The Australian National University, Oct 1989.

[Bre89] D.M. Bressoud, "Factorisation and Primality Testing," Undergraduate Texts in Mathematics, Springer-Verlag, New York, 1989.

[Bri88] J. Brillhart, D.H. Lehmer, J.L. Selfridge, B. Tuckerman and S.S. Wagstaff Jr., "Factorizations of $b^n \pm 1$, b=2,3,5,6,7,10,11,12 up to High Powers," Volume 22 of Contemporary Mathematics, American Mathematical Society, Providence, Rhode Island, 2nd Ed., 1988.

[BLZ94] J. Buchmann, J. Loho and J. Zayer, "An Implementation of the General Number Field Sieve," Advances in Cryptology - Crypto '93, Springer-Verlag, New York, 1994.

[Buh92] J.P. Buhler, H.W. Lenstra and C. Pomerance, "Factoring Integers with the Number Field Sieve," 1992.

[Cor90] T.H. Cormen, C.E. Leiserson and R.L. Rivest, "An Introduction to Algorithms," MIT Press, Cambridge, Massachusetts, 1990.

[Dix91] Dixon, B., Lenstra. A.K., "Massively Parallel Elliptic Curve Factoring," 1991.

[Fah94] Fahn, Paul, "FAQ about today's Cryptography," pp. 5. Available from: http://www.rsa.com/rsalabs/faq/StartHere.html

[Koy91] Yoichi Koyama, available from: http://alpha.acast.nova.edu/simtel/ubasic.html.

[Len87] H.W. Lenstra Jr., “Factoring Integers with Elliptic Curves,” Ann. of Math., 126:649-673, 1987.

[Len90a] A.K. Lenstra and M.S. Manasse, “Factoring by Electronic Mail,” Advances in Cryptology - Eurocrypt ‘89 Proceedings, Berlin, Springer-Verlag, 1990, pp. 355-371.

[Len90b] A.K. Lenstra, H.W. Lenstra Jr., M.S. Manasse and J.M. Pollard, “The Number Field Sieve,” Proceeding of the 22nd ACM Symposium on the Theory of Computing, 1990, pp. 564 - 572.

[Len91] A.K. Lenstra and M.S. Manasse, “Factoring with Two Large Primes,” Advances in Cryptology - Eurocrypt ‘90, page 72-82, Springer-Verlag, Berlin, 1991.

[Len93] A.K. Lenstra, H.W. Lenstra Jr., M.S. Manasse, and M.S. Pollard, “The Factorisation of the Ninth Fermat Number,” Mathematics of Computation, vol. 61, num. 203, Jul 1993, pp. 319-350.

[Mil86] V.S. Miller, “Use of Elliptic Curves in Cryptography,” Advances in Cryptology - Crypto ‘85, pages 417-426, Springer-Verlag, New-York, 1986.

[Mon87] P. Montgomery, “Speeding the Pollard and Elliptic Curves Methods,” Math. Comp., 48, Jan. 1987, pp 243-364.

[Pol74] J. Pollard, “Theorems of Factorisation and Primality Testing,” Proc. Cambridge Philos. Soc., 76:521-528, 1974.

[Pol75] J. Pollard, “Monte Carlo Method for Factorisation,” BIT, 15:331-334, 1975.

[Sch94] B. Schneier, “Applied Cryptography - Protocols, Algorithms, and Source Code in C,” Wiley, 1994.

[Sco89] M. Scott, “Factoring Large Integers on Small Computers,” Working Paper: CA-0169, School of Computer Applications, Dublin City University, Dublin, Ireland.

[Sco95] M. Scott, The MIRACL Programming Library, available from: ftp://ftp.compapp.dcu.ie/pub/crypto/.

[Sil87] R.D. Silverman., “The Multiple Polynomial Quadratic Sieve,” Math. Comp, 48:329-339, 1987.

[Wil82] H.C. Williams, “A p+1 method of factoring,” Math. Comp., 39:225-234, 1982.

7. Results

7.1 The Environment Used

The test environment consisted of an ethernet network of IBM RS/6000's running AIX 3.2, IBM's flavour of Unix. This platform provides full multi-tasking, as well as a local and remote inter-process communications facility.

The maximum number of workstations used at any one time was 17. The code was compiled with xlC - IBM's C++ compiler for AIX, and a special numerical library for handling very large numbers and arithmetic - MIRACL [Sco95], was used in the implementation of the ECM factoring program.

It should be noted that the tests were carried out while other users used the network and workstations although the load was relatively light.

7.2 Notes on the ECM Factoring Program

The Elliptic Curve Factoring program used was adapted from [Sco89] for use in a parallel system. No major changes to the basic ECM code were required, however the formulae to calculate the limits was updated. The new formula to calculate the optimum limits in the ECM program are from the ECM factoring program included in Ubasic [Koy91].

i.e.,

Limit1 = int(log(N)^2.65/10)

Limit1 = min(130000, max(LIMIT1,500))

Limit2 = 40*Limit1

7.3 Performance of the DistProc System

Utilising between one and seventeen DPSlaves, two numbers - 10^{39}+1 and 10^{59}+1, were factored using the distributed version of Lenstra's Elliptic Curve factoring method described in section 6.3.1. The following is a summary of the results presented at the end of this chapter.

7.3.1 Factorisation of $10^{39} + 1$

The following figures are averaged from several tests and represent a good cross section of typical results. The time required for each process (each process represents one curve in an ECM factorisation) to complete was 88.1 seconds on average and overall there were 121 processes tried (in this example). There was only one remote process (curve) per DISTPROC Slave at any one time and each DISTPROC Slave was running on a separate computer, although other users were using the computers and network while the tests ran. The results presented in [Table 1] and the graphs that follow are based on the timings and results shown in section 7. These results were obtained from the factorisation of $10^{37} + 1$ using a distributed version of Lenstra's Elliptic Curve factorisation method.

Number of DPSlaves (IBM RS/6000 320's) used	Run Time with Security (Mins)	Curves / minute with Security	Speedup over 1 DISTPROC Slave
1	164.65	0.73	1.00
2	81.87	1.48	2.01
4	44.43	2.72	3.71
8	22.93	5.28	7.18
16	12.30	9.84	13.39

Table 1. Times and speedup for the factorisation of $10^{37}+1$ using a distributed version of Lenstra's Elliptic Curve factoring method.

The greater than linear speedup achieved when going from one to two DPSlaves could have arisen from a change in the load on the network or from other users using one of the DPSlave computers. While this is unfortunate it is an acceptable aspect of distributed computing when using a network of shared computers.

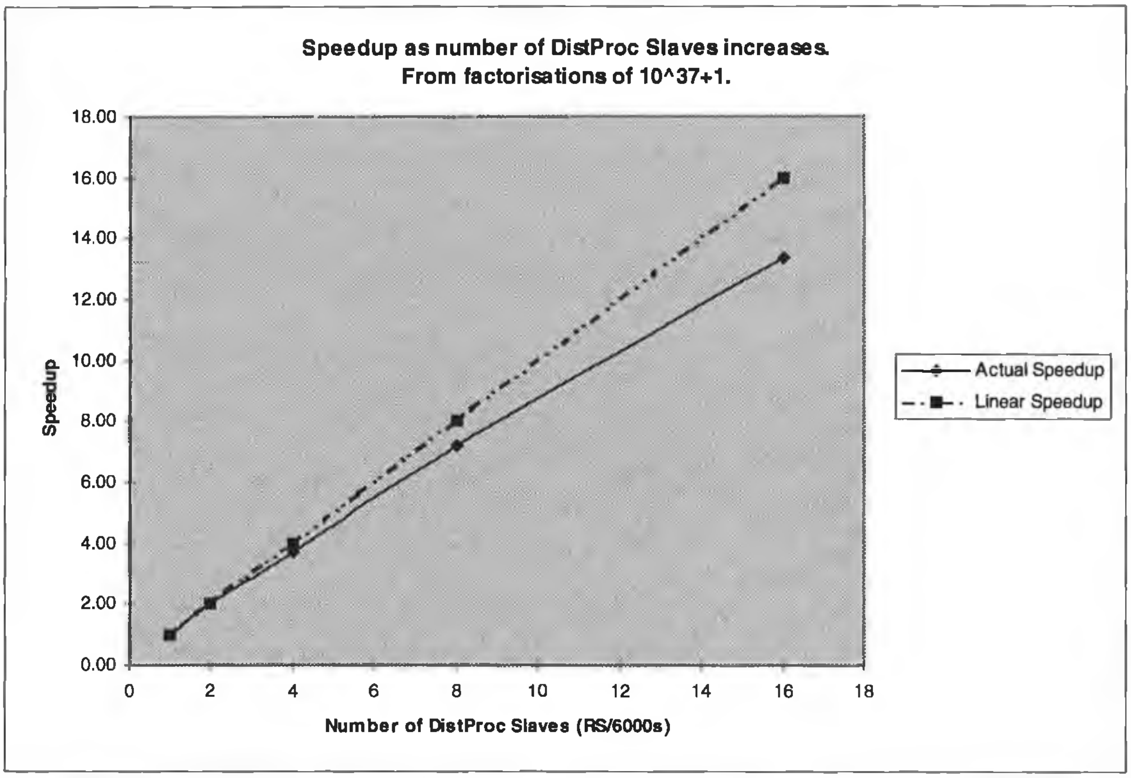


Figure 17. Graph of processing speedup in the DISTPROC System.

The graph in [Figure 17] is based on the figures in [Table 1] and compares the maximum possible speedup (linear speedup) against the actual speedup achieved. It should be noted that these measurements are from a suitable application where there is a high ratio of computation to communication. As the level of communication increases the actual speedup would fall.

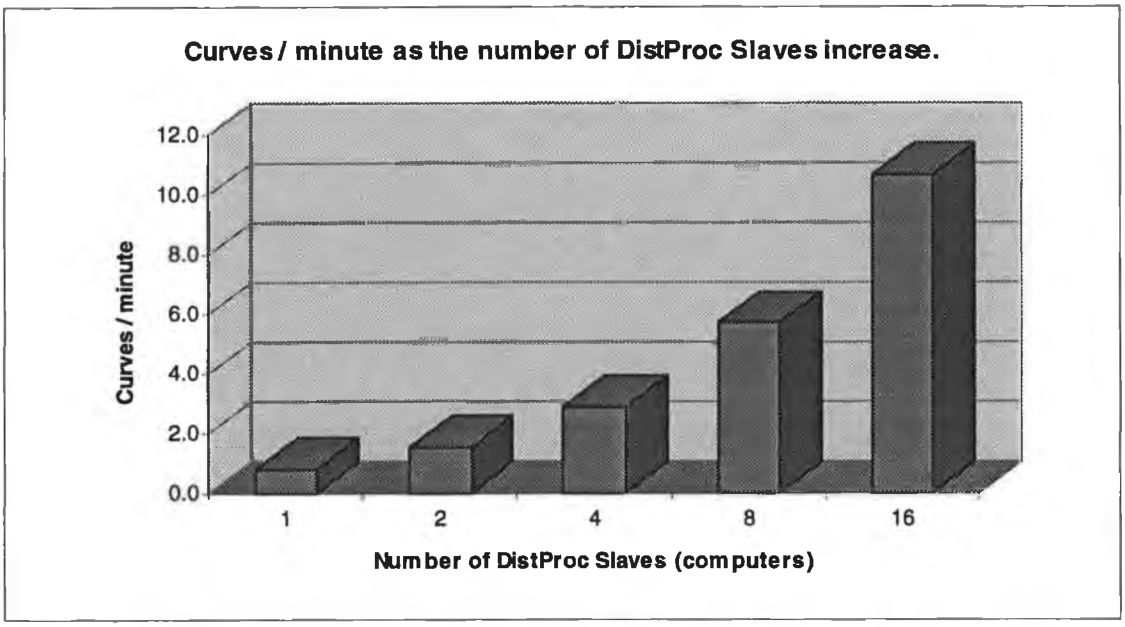


Figure 18. Graph of curves/min as the number of DISTPROC Slaves increase.

The graph in [Figure 18] shows how the number of curves (an ECM factorisation tries many curves when trying to factor a number) completed per minute increases as the number of DISTPROC Slaves increases up to a maximum of 16.

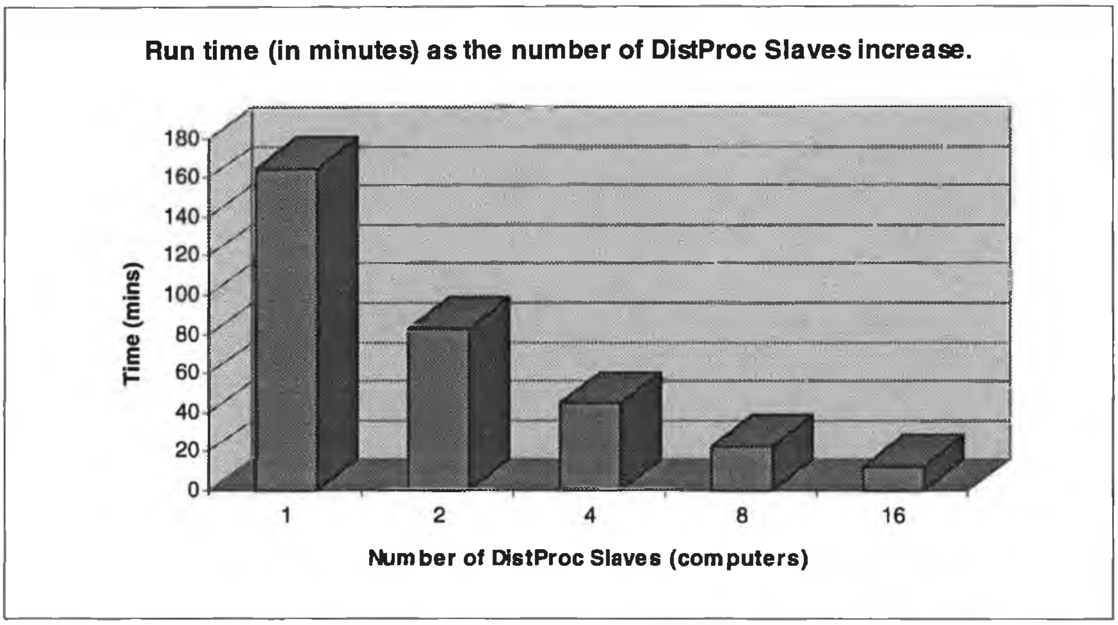


Figure 19. Graph of DISTPROC Slaves against run time (mins)

The graph in [Figure 19] shows how the time taken in minutes to complete a set amount of work decreases as the number of DISTPROC Slaves increases to a maximum of 16.

7.3.2 Factorisation of $10^{59} + 1$

The results from the factorisation of 10^{59}+1 using 17 DPSlaves are shown in section 7.7.1. The important information from this set of results is:

```
Start Time: Sun Jan 15 01:07:50 1995
Trying to factor ((10^59)+1)
Factors of 100000000000000000000000000000000000000000000000000000000001 = (10^59)+1
found with curve #253
Limit1 = 36929 and Limit2 = 1477160
prime factors 1090805842068098677837
prime factor (num = num/t) 4411922770996074109644535362851087
Start Time: Sun Jan 15 01:07:50 1995
Finish Time: Sun Jan 15 06:04:37 1995
Running Time: 296.783 minutes.
There were 17 DPSlaves used.
```

The results from this factorisation cannot be compared directly to the results obtained from the factorisation of 10^{37}+1 since completely different limits and curves were used during the factorisation process.

7.4 Security

The DISTPROC System is a non-dedicated distributed parallel processing system in that it runs on resources which are shared with other users. Currently the balancing algorithm does not distinguish between processes from different DISTPROC Client applications, which could result in one application hogging the system if it gets in first.

If a DISTPROC system (cluster) is shared between several users then security must be used in order to authenticate each user.

The DISTPROC system was tested with and without security. All told, the overhead for security added on average two seconds to the setup time of each remote process. The overhead of using security varies between different systems but is generally negligible in comparison to the computation time of typical DISTPROC applications. However most systems will not be concerned with security and will therefore not encounter any time penalties.

7.5 Explanation of the Results

In the result output the following should be noted that all times were calculated on the DPClient and that wait requests wait on specific processes to finish - not the first available process.

Another point is that the DPClient is responsible for buffering run process requests due to a bug in the DPServer's load balancing mechanism. Once a wait request completes the DPClient starts the next process running. This can result in a wait request on a slow process delaying the execution of the next process, and even if another DPSlave becomes available in the mean time (by a process finishing while a wait request is blocked) it will not receive a process until the blocked wait request completes.

There are many reasons that one process could take longer to complete than other processes, for instance if a computer is heavily loaded by other jobs or is simply not as powerful a computer as the other computers in the DISTPROC system.

The following explains the meaning of the columns in the results output.

The *Start Time* is when the DPClient application issued a run request to a DPServer.

End Time is when the DPClient application completed a wait() call. Wait requests were paired with an associated run request and the DPClient was responsible for process buffering which explains why there is a knock on effect once one process is delayed.

This effect shows up clearly under *Wait Time* which is the time that the DPClient was blocked waiting for a process to finish.

Run Setup Time gives the time taken to start a process running. This covers the time taken for a DPClient to send a run request to a DPServer and get a process id back. This column shows the additional setup time overhead that enabling security incurs.

7.6 Results from the Factorisation of 10^37 + 1

7.6.1 Results for One DPSlave with Security

```
Start Time:Thu Jan 12 21:37:39 1995
Trying to factor ((10^37)+1)
Received result from RPLenstra #121
Factors of 10000000000000000000000000000000000000001 = (10^37)+1
found with curve #124
Limit1 = 9037 and Limit2 = 361480
prime factors 422650073734453
prime factor (num = num/t) 296557347313446299
Start Time: Thu Jan 12 21:37:39 1995
Finish Time: Fri Jan 13 00:22:18 1995
Running Time: 164.65 minutes.
There were 1 DPSlaves used.

curve   (Start Time)   (End Time)   (Run Setup Time)   (Wait Time)
3        789946660      789946742       4        78
4        789946742      789946823       3        78
5        789946823      789946904       3        78
6        789946904      789946985       3        78
7        789946985      789947066       3        78
8        789947066      789947147       3        78
9        789947147      789947228       3        78
10       789947228      789947310       4        78
11       789947310      789947391       3        78
12       789947391      789947472       3        78
13       789947472      789947553       3        78
14       789947553      789947634       3        78
15       789947634      789947715       3        78
16       789947715      789947797       3        79
17       789947801      789947883       4        78
18       789947883      789947964       3        78
19       789947964      789948045       3        78
20       789948045      789948126       3        78
21       789948126      789948208       4        78
22       789948208      789948289       3        78
23       789948289      789948369       3        77
24       789948370      789948451       2        79
25       789948451      789948532       4        77
26       789948532      789948613       4        77
27       789948613      789948694       4        77
28       789948694      789948776       4        78
29       789948776      789948857       3        78
30       789948857      789948938       3        78
31       789948938      789949019       3        78
32       789949023      789949104       3        78
33       789949104      789949187       5        78
34       789949187      789949267       3        77
35       789949267      789949348       4        77
36       789949348      789949430       3        79
37       789949430      789949511       3        78
38       789949511      789949592       3        78
39       789949592      789949673       3        78
40       789949673      789949755       4        78
41       789949755      789949836       3        78
42       789949836      789949917       3        78
43       789949917      789949998       3        78
44       789949998      789950079       3        78
45       789950079      789950161       4        78
46       789950161      789950242       3        78
47       789950242      789950323       3        78
48       789950323      789950404       3        78
49       789950404      789950485       3        78
50       789950485      789950566       3        78
51       789950566      789950648       4        78
52       789950648      789950729       3        78
53       789950729      789950810       3        78
54       789950810      789950891       4        77
55       789950891      789950972       4        77
56       789950972      789951054       4        78
57       789951054      789951135       3        78
58       789951135      789951215       3        77
59       789951215      789951297       4        78
60       789951297      789951378       3        78
61       789951378      789951459       3        78
62       789951464      789951545       3        78
63       789951545      789951626       3        78
64       789951626      789951707       3        78
65       789951707      789951788       3        78
66       789951788      789951869       3        78
67       789951869      789951950       3        78
68       789951950      789952031       3        78
69       789952031      789952113       4        78
70       789952113      789952194       3        78
71       789952194      789952274       3        77
72       789952274      789952356       3        79
73       789952356      789952437       3        78
74       789952437      789952518       3        78
75       789952518      789952599       3        78
76       789952599      789952680       3        78
77       789952685      789952766       3        78
78       789952766      789952847       3        78
79       789952847      789952928       3        78
80       789952928      789953009       3        78
81       789953009      789953091       4        78
82       789953091      789953172       3        78
83       789953172      789953253       3        78
84       789953253      789953334       3        78
85       789953334      789953415       3        78
86       789953415      789953496       3        78
87       789953496      789953577       3        78
88       789953577      789953658       3        78
89       789953658      789953740       4        78
90       789953740      789953821       3        78
91       789953821      789953902       3        78
92       789953902      789953983       3        78
93       789953983      789954064       3        78
94       789954064      789954145       3        78
95       789954145      789954226       3        78
96       789954226      789954307       3        78
97       789954307      789954389       4        78
98       789954389      789954470       3        78
99       789954470      789954551       3        78
100      789954551      789954632       3        78
101      789954632      789954713       3        78
102      789954713      789954794       3        78
103      789954794      789954875       3        78
104      789954875      789954957       3        79
105      789954957      789955038       3        78
106      789955038      789955119       3        78
107      789955123      789955204       3        78
108      789955204      789955285       3        78
109      789955285      789955367       3        79
110      789955367      789955447       3        77
111      789955447      789955529       4        78
112      789955529      789955610       3        78
113      789955610      789955692       4        78
114      789955692      789955772       3        77
115      789955772      789955853       4        77
116      789955853      789955935       4        78
117      789955935      789956016       3        78
118      789956016      789956097       3        78
119      789956097      789956179       4        78
120      789956179      789956260       3        78
121      789956260      789956341       3        78
122      789956341      789956422       3        78
123      789956422      789956503       3        78
124      789956503      789956537       3        31
```

7.6.2 Results for Two DPSlaves with Security

```
Start Time:Thu Jan 12 17:27:23 1995
Trying to factor ((10^37)+1)
Received result from RPLenstra #121
Factors of 10000000000000000000000000000000000001 = (10^37)+1
found with curve #124
Limit1 = 9037 and Limit2 = 361480
prime factors 422650073734453
prime factor (num = num/t) 2965573473134462 99
Start Time: Thu Jan 12 17:27:23 1995
Finish Time: Thu Jan 12 18:49:15 1995
Running Time: 81.8667 minutes.
There were 2 DPSlaves used.

curve  (Start Time)  (End Time)  (Run Setup Time)  (Wait Time)
3       789931644     789931714     3      63
4       789931647     789931729     4      13
5       789931714     789931782     2      50
6       789931729     789931810     3      25
7       789931782     789931850     3      37
8       789931810     789931891     3      38
9       789931850     789931920     3      25
10      789931891     789931973     4      50
11      789931920     789931989     3      13
12      789931973     789932054     3      63
13      789931989     789932057     2      0
14      789932054     789932135     3      76
15      789932057     789932138     2      0
16      789932135     789932216     3      76
17      789932138     789932220     2      0
18      789932216     789932298     4      76
19      789932220     789932301     2      0
20      789932298     789932379     3      71
21      789932301     789932383     7      0
22      789932379     789932460     4      74
23      789932383     789932464     3      0
24      789932460     789932542     4      76
25      789932464     789932547     2      1
26      789932542     789932624     4      74
27      789932547     789932628     3      0
28      789932625     789932706     3      75
29      789932628     789932709     3      0
30      789932706     789932787     3      75
31      789932709     789932790     3      0
32      789932787     789932869     3      77
33      789932790     789932873     2      0
34      789932869     789932951     4      76
35      789932873     789932954     2      0
36      789932951     789933032     3      75
37      789932954     789933035     3      0
38      789933032     789933112     3      75
39      789933035     789933116     2      0
40      789933112     789933194     4      75
41      789933116     789933198     3      1
42      789933194     789933276     3      76
43      789933198     789933279     2      0
44      789933276     789933357     3      75
45      789933279     789933360     3      0
46      789933357     789933437     3      74
47      789933360     789933447     3      1
48      789933442     789933525     4      74
49      789933447     789933537     4      8
50      789933525     789933608     4      66
51      789933537     789933667     5      43
52      789933608     789933704     16     33
53      789933667     789933744     4      37
54      789933704     789933785     3      39
55      789933744     789933817     2      29
56      789933785     789933867     3      46
57      789933817     789933888     4      18
58      789933867     789933948     3      57
59      789933888     789933957     3      6
60      789933948     789934029     3      68
61      789933957     789934033     4      1
62      789934029     789934110     3      75
63      789934033     789934113     2      0
64      789934110     789934191     3      74
65      789934113     789934195     4      0
66      789934191     789934273     4      76
67      789934195     789934278     2      0
68      789934273     789934356     5      75
69      789934278     789934359     3      0
70      789934356     789934437     3      76
71      789934359     789934440     2      0
72      789934437     789934518     3      75
73      789934440     789934522     3      0
74      789934518     789934600     4      76
75      789934522     789934603     2      0
76      789934600     789934680     3      75
77      789934603     789934684     2      0
78      789934680     789934762     4      76
79      789934684     789934765     2      0
80      789934762     789934843     3      76
81      789934765     789934847     2      0
82      789934844     789934925     3      75
83      789934847     789934928     3      0
84      789934925     789935006     3      74
85      789934928     789935009     4      0
86      789935006     789935087     3      76
87      789935009     789935090     2      0
88      789935087     789935168     3      76
89      789935090     789935172     2      1
90      789935168     789935249     3      75
91      789935172     789935253     2      1
92      789935249     789935330     3      75
93      789935253     789935334     2      0
94      789935330     789935411     4      75
95      789935334     789935415     2      1
96      789935411     789935492     3      75
97      789935415     789935511     2      14
98      789935493     789935575     4      60
99      789935511     789935576     4      0
100     789935575     789935653     1      74
101     789935576     789935657     3      3
102     789935653     789935720     1      60
103     789935657     789935738     3      17
104     789935720     789935789     1      48
105     789935738     789935819     3      24
106     789935794     789935871     1      48
107     789935819     789935901     4      28
108     789935871     789935939     2      35
109     789935901     789935982     3      42
110     789935939     789936007     1      22
111     789935982     789936063     3      56
112     789936007     789936073     0      7
113     789936063     789936144     3      70
114     789936073     789936148     1      0
115     789936144     789936226     4      77
116     789936148     789936229     1      0
117     789936226     789936307     3      77
118     789936229     789936310     1      0
119     789936307     789936388     3      77
120     789936310     789936391     1      0
121     789936388     789936470     3      78
122     789936391     789936473     1      0
123     789936470     789936551     3      77
124     789936473     789936555     1      1
```

7.6.3 Results for Four DPSlaves with Security

```
Start Time:Fri Jan 13 15:25:25 1995
Trying to factor ((10^37)+1)
Received result from RPLenstra #121
Factors of 10000000000000000000000000000000000001 = (10^37)+1
found with curve #124
Limit1 = 9037 and Limit2 = 361480
prime factors 422650073734453
prime factor (num = num/t) 296557347313446299
Start Time: Fri Jan 13 15:25:25 1995
Finish Time: Fri Jan 13 16:09:51 1995
Running Time: 44.4333 minutes.
There were 4 DPSlaves used.

curve  (Start Time)  (End Time)  (Run Setup Time)  (Wait Time)
3       790010726     790010805     1     72
4       790010727     790010822     4     13
5       790010731     790010823     1     0
6       790010732     790010825     1     1
7       790010805     790010887     4     61
8       790010822     790010911     1     23
9       790010823     790010913     1     1
10      790010825     790010914     1     0
11      790010887     790010966     1     51
12      790010911     790010997     1     30
13      790010913     790010998     1     0
14      790010914     790011000     1     0
15      790010966     790011045     1     44
16      790010997     790011080     1     34
17      790010998     790011082     2     0
18      790011000     790011091     1     8
19      790011045     790011123     1     31
20      790011080     790011166     2     41
21      790011082     790011167     1     0
22      790011091     790011177     1     9
23      790011123     790011202     2     24
24      790011166     790011258     1     55
25      790011168     790011260     0     1
26      790011177     790011261     1     0
27      790011202     790011281     1     18
28      790011258     790011345     1     63
29      790011260     790011347     1     0
30      790011261     790011348     2     0
31      790011281     790011360     1     11
32      790011345     790011431     2     70
33      790011347     790011437     1     0
34      790011348     790011438     1     0
35      790011360     790011439     1     0
36      790011435     790011525     2     85
37      790011437     790011527     1     0
38      790011438     790011528     1     0
39      790011439     790011529     1     0
40      790011525     790011614     2     84
41      790011527     790011616     1     0
42      790011528     790011617     1     0
43      790011529     790011618     1     0
44      790011614     790011704     2     85
45      790011616     790011706     1     0
46      790011617     790011707     1     0
47      790011618     790011709     1     0
48      790011704     790011793     2     83
49      790011706     790011794     1     0
50      790011707     790011795     2     0
51      790011709     790011797     1     0
52      790011793     790011883     1     85
53      790011794     790011885     1     1
54      790011795     790011886     2     0
55      790011797     790011887     1     0
56      790011883     790011971     1     82
57      790011885     790011973     1     1
58      790011886     790011974     1     0
59      790011887     790011975     2     0
60      790011971     790012056     1     80
61      790011973     790012058     1     1
62      790011974     790012059     1     0
63      790011975     790012060     1     0
64      790012056     790012146     1     85
65      790012058     790012147     1     0
66      790012059     790012148     1     0
67      790012060     790012150     1     0
68      790012146     790012232     1     81
69      790012147     790012234     1     0
70      790012148     790012235     2     0
71      790012150     790012236     1     0
72      790012232     790012320     2     83
73      790012234     790012322     1     0
74      790012235     790012323     1     0
75      790012236     790012325     1     1
76      790012321     790012408     1     83
77      790012322     790012410     1     0
78      790012323     790012411     1     0
79      790012325     790012413     0     1
80      790012408     790012495     2     81
81      790012410     790012496     1     0
82      790012411     790012498     1     0
83      790012413     790012499     1     0
84      790012495     790012581     1     80
85      790012496     790012583     2     0
86      790012498     790012584     1     0
87      790012499     790012585     2     0
88      790012581     790012670     2     84
89      790012583     790012672     1     1
90      790012584     790012673     1     1
91      790012585     790012674     1     0
92      790012670     790012755     1     80
93      790012672     790012757     0     0
94      790012673     790012758     1     0
95      790012674     790012760     1     0
96      790012755     790012845     2     84
97      790012757     790012846     1     0
98      790012758     790012848     2     1
99      790012760     790012849     1     1
100     790012845     790012929     1     79
101     790012847     790012931     0     0
102     790012848     790012932     0     1
103     790012849     790012933     1     0
104     790012930     790013015     1     81
105     790012931     790013016     0     0
106     790012932     790013017     1     0
107     790012933     790013019     1     1
108     790013015     790013099     1     79
109     790013016     790013100     1     0
110     790013017     790013101     1     0
111     790013019     790013103     1     1
112     790013099     790013183     1     79
113     790013100     790013185     1     0
114     790013101     790013186     1     0
115     790013103     790013187     1     0
116     790013183     790013268     2     80
117     790013185     790013269     1     0
118     790013186     790013270     1     0
119     790013187     790013272     1     0
120     790013268     790013353     1     80
121     790013269     790013354     1     0
122     790013270     790013355     2     0
123     790013272     790013356     1     0
124     790013353     790013390     1     33
```

7.6.4 Results for Eight DPSlaves with Security

```
Trying to factor ((10^37)+1)
Received result from RPLenstra #121
Factors of 10000000000000000000000000000000000001 = (10^37)+1
found with curve #124
Limit1 = 9037 and Limit2 = 361480
prime factors 422650073734453
prime factor (num = num/t) 296557347313446299
Start Time: Tue Jan 10 22:20:24 1995
Finish Time: Tue Jan 10 22:43:20 1995
Running Time: 22.9333 minutes.
There were 8 DPSlaves used.

curve  (Start Time)   (End Time)   (Run Setup Time)   (Wait Time)
3       789776425      789776512        4        55
4       789776429      789776515        3        0
5       789776432      789776518        3        0
6       789776435      789776523        6        1
7       789776441      789776528        3        1
8       789776444      789776531        3        0
9       789776447      789776534        4        0
10      789776451      789776537        6        0
11      789776512      789776598        3        58
12      789776515      789776601        3        0
13      789776518      789776604        4        1
14      789776523      789776611        4        4
15      789776528      789776615        3        0
16      789776531      789776618        3        0
17      789776534      789776621        3        0
18      789776537      789776624        3        0
19      789776598      789776684        3        56
20      789776601      789776687        2        0
21      789776604      789776690        3        1
22      789776611      789776698        4        6
23      789776615      789776701        3        0
24      789776618      789776705        3        1
25      789776621      789776708        3        1
26      789776624      789776712        4        0
27      789776684      789776769        3        54
28      789776687      789776773        2        1
29      789776690      789776776        2        0
30      789776698      789776784        3        5
31      789776701      789776787        3        0
32      789776705      789776791        2        0
33      789776708      789776794        4        0
34      789776712      789776798        3        0
35      789776769      789776855        3        54
36      789776773      789776858        3        0
37      789776776      789776860        3        0
38      789776784      789776870        3        7
39      789776787      789776873        4        0
40      789776791      789776876        3        0
41      789776794      789776880        4        0
42      789776798      789776883        3        0
43      789776855      789776941        3        55
44      789776858      789776944        2        0
45      789776860      789776946        3        0
46      789776870      789776956        3        7
47      789776873      789776959        3        0
48      789776876      789776962        4        0
49      789776880      789776966        3        0
50      789776883      789776969        3        0
51      789776941      789777026        3        54
52      789776944      789777029        2        0
53      789776946      789777033        3        1
54      789776956      789777042        3        6
55      789776959      789777045        3        0
56      789776962      789777048        4        0
57      789776966      789777051        3        0
58      789776969      789777055        3        0
59      789777026      789777112        3        54
60      789777029      789777115        3        0
61      789777033      789777117        3        0
62      789777042      789777128        3        7
63      789777045      789777131        3        0
64      789777048      789777135        3        0
65      789777051      789777138        4        0
66      789777055      789777142        3        1
67      789777112      789777197        3        53
68      789777115      789777201        2        1
69      789777117      789777203        4        0
70      789777128      789777214        3        8
71      789777131      789777217        4        0
72      789777135      789777221        3        1
73      789777138      789777224        3        0
74      789777142      789777227        2        0
75      789777197      789777283        3        52
76      789777201      789777286        2        0
77      789777203      789777289        3        0
78      789777214      789777300        3        7
79      789777217      789777303        3        0
80      789777221      789777307        3        1
81      789777224      789777310        3        0
82      789777227      789777313        4        0
83      789777283      789777369        3        53
84      789777286      789777372        3        0
85      789777289      789777375        4        0
86      789777300      789777386        3        8
87      789777303      789777390        3        1
88      789777307      789777393        3        1
89      789777310      789777396        3        0
90      789777313      789777399        3        0
91      789777369      789777454        3        52
92      789777372      789777458        3        1
93      789777375      789777461        3        0
94      789777386      789777472        3        8
95      789777390      789777476        2        0
96      789777393      789777479        3        0
97      789777396      789777483        3        1
98      789777399      789777487        3        1
99      789777454      789777540        3        51
100     789777458      789777543        3        0
101     789777461      789777545        3        0
102     789777472      789777559        4        10
103     789777476      789777562        3        0
104     789777479      789777565        3        0
105     789777483      789777569        3        0
106     789777487      789777572        2        0
107     789777540      789777625        3        50
108     789777543      789777628        2        0
109     789777545      789777631        4        1
110     789777559      789777645        3        11
111     789777562      789777648        3        0
112     789777565      789777651        4        0
113     789777569      789777655        3        1
114     789777572      789777658        3        0
115     789777625      789777711        3        50
116     789777628      789777714        2        0
117     789777631      789777717        3        0
118     789777645      789777731        3        11
119     789777648      789777735        3        1
120     789777651      789777738        3        0
121     789777655      789777742        3        1
122     789777658      789777745        3        1
123     789777711      789777796        3        48
124     789777714      789777800        3        1
```

7.6.5 Results for Sixteen DPSlaves with Security

```
Start Time:Tue Jan 10 23:17:53 1995
 Trying to factor ((10^37)+1)
 Received result from RPLenstra #121
 Factors of 10000000000000000000000000000000000001 = (10^37)+1
 found with curve #124
Limit1 = 9037 and Limit2 = 361480
 prime factors 4226500733734453
prime factor (num = num/t) 2965573473134462 99
 Start Time: Tue Jan 10 23:17:53 1995
Finish Time: Tue Jan 10 23:30:11 1995
Running Time: 12.3 minutes.
There were 16 DPSlaves used.

curve  (Start Time)  (End Time)  (Run Setup Time)  (Wait Time)
 3      789779874     789779946     2     14
4      789779876     789779963     4     14
5      789779880     789779966     3     0
6      789779883     789779969     3     0
7      789779886     789779972     3     0
8      789779889     789779978     4     3
9      789779893     789779981     3     0
10     789779896     789779985     6     0
11     789779902     789779989     4     0
12     789779906     789779993     3     1
13     789779909     789779996     3     1
14     789779912     789779999     4     1
15     789779916     789780003     4     1
16     789779920     789780007     4     1
17     789779924     789780012     4     1
18     789779928     789780015     4     0
19     789779946     789780019     3     0
20     789779963     789780050     3     29
21     789779966     789780056     3     3
22     789779969     789780060     3     1
23     789779972     789780063     3     1
24     789779978     789780066     3     1
25     789779982     789780070     3     0
26     789779986     789780073     3     0
27     789779989     789780076     3     0
28     789779993     789780079     2     0
29     789779996     789780083     2     0
30     789779999     789780086     3     0
31     789780003     789780089     3     0
32     789780007     789780094     4     2
33     789780012     789780097     3     0
34     789780015     789780101     4     1
35     789780019     789780104     2     0
36     789780050     789780135     3     29
37     789780057     789780142     2     3
38     789780060     789780146     2     0
39     789780063     789780150     2     1
40     789780066     789780155     4     3
41     789780070     789780158     3     0
42     789780073     789780161     3     0
43     789780076     789780164     3     0
44     789780079     789780167     4     0
45     789780083     789780171     3     1
46     789780086     789780174     3     1
47     789780089     789780177     3     0
48     789780094     789780180     3     0
49     789780097     789780183     3     0
50     789780101     789780187     3     1
51     789780104     789780190     2     1
52     789780135     789780221     4     30
53     789780142     789780229     4     5
54     789780146     789780232     3     0
55     789780150     789780235     2     0
56     789780155     789780243     3     5
57     789780158     789780246     3     0
58     789780161     789780249     3     0
59     789780164     789780252     3     0
60     789780167     789780255     3     0
61     789780171     789780258     2     0
62     789780174     789780261     3     0
63     789780177     789780264     3     0
64     789780180     789780268     3     1
65     789780183     789780272     3     0
66     789780187     789780275     2     0
67     789780190     789780278     1     0
68     789780221     789780307     3     27
69     789780229     789780314     3     4
70     789780232     789780317     3     0
71     789780235     789780321     3     1
72     789780243     789780331     3     7
73     789780246     789780335     3     0
74     789780249     789780339     3     1
75     789780252     789780342     3     1
76     789780255     789780345     3     0
77     789780258     789780348     3     1
78     789780261     789780351     3     1
79     789780264     789780354     3     1
80     789780268     789780357     4     0
81     789780272     789780360     3     0
82     789780275     789780363     3     0
83     789780278     789780367     2     0
84     789780307     789780393     3     24
85     789780314     789780400     3     4
86     789780317     789780403     3     0
87     789780321     789780406     3     0
88     789780331     789780421     4     12
89     789780335     789780425     3     1
90     789780339     789780429     2     0
91     789780342     789780432     3     0
92     789780345     789780435     2     0
93     789780348     789780438     2     0
94     789780351     789780441     2     0
95     789780354     789780444     3     0
96     789780357     789780447     3     0
97     789780360     789780450     3     0
98     789780363     789780453     4     0
99     789780367     789780456     2     0
100    789780393     789780478     3     20
101    789780400     789780485     3     3
102    789780403     789780489     3     1
103    789780406     789780493     3     0
104    789780421     789780509     3     13
105    789780425     789780513     4     1
106    789780429     789780516     3     1
107    789780432     789780518     3     0
108    789780435     789780522     3     1
109    789780438     789780526     3     0
110    789780441     789780529     3     0
111    789780444     789780532     3     0
112    789780447     789780536     3     1
113    789780450     789780539     3     1
114    789780453     789780541     3     0
115    789780456     789780545     2     1
116    789780478     789780565     4     19
117    789780485     789780571     3     3
118    789780489     789780574     4     0
119    789780493     789780578     3     1
120    789780509     789780597     3     17
121    789780513     789780601     2     1
122    789780516     789780604     2     0
123    789780519     789780607     2     0
124    789780522     789780610     4     0
```

7.6.6 Results for One DPSlave without Security

```
Start Time:Fri Jan 13 09:16:26 1995
Trying to factor ((10^37)+1)
Received result from RPLenstra #121
Factors of 10000000000000000000000000000000000001 = (10^37)+1
found with curve #124
Limit1 = 9037 and Limit2 = 361480
prime factors 422650073734453
prime factor (num = num/t) 296557347313446299
Start Time: Fri Jan 13 09:16:26 1995
Finish Time: Fri Jan 13 11:56:45 1995
Running Time: 160.317 minutes.
There were 1 DPSlaves used.


curve  (Start Time)   (End Time)   (Run Setup Time)  (Wait Time)
3        789988587      789988666       1        78
4        789988666      789988744       1        77
5        789988744      789988824       2        78
6        789988824      789988905       1        80
7        789988905      789988984       1        78
8        789988984      789989062       1        77
9        789989062      789989141       2        77
10       789989141      789989220       1        78
11       789989220      789989299       2        77
12       789989299      789989378       1        78
13       789989378      789989457       1        78
14       789989457      789989536       1        78
15       789989536      789989615       1        78
16       789989615      789989693       1        77
17       789989693      789989772       2        77
18       789989772      789989851       1        78
19       789989851      789989930       1        78
20       789989930      789990009       1        78
21       789990009      789990088       1        78
22       789990088      789990166       1        77
23       789990166      789990245       2        77
24       789990245      789990324       1        78
25       789990324      789990403       1        78
26       789990403      789990482       1        78
27       789990482      789990560       1        77
28       789990560      789990639       2        77
29       789990639      789990718       2        77
30       789990723      789990802       1        78
31       789990802      789990880       1        77
32       789990881      789990960       1        78
33       789990960      789991039       1        78
34       789991039      789991118       1        78
35       789991118      789991197       1        78
36       789991197      789991280       1        82
37       789991280      789991368       2        86
38       789991368      789991446       1        77
39       789991446      789991525       2        77
40       789991525      789991605       2        78
41       789991605      789991687       4        78
42       789991687      789991766       1        78
43       789991766      789991844       1        77
44       789991844      789991923       2        77
45       789991928      789992006       1        77
46       789992006      789992085       2        77
47       789992085      789992164       1        78
48       789992164      789992243       1        78
49       789992243      789992321       1        77
50       789992321      789992400       1        78
51       789992400      789992478       1        77
52       789992478      789992557       1        78
53       789992557      789992636       2        77
54       789992636      789992714       1        77
55       789992714      789992793       2        77
56       789992793      789992872       3        77
57       789992872      789992951       1        78
58       789992951      789993030       1        78
59       789993030      789993108       1        77
60       789993113      789993191       1        77
61       789993191      789993270       2        77
62       789993270      789993350       2        78
63       789993350      789993429       1        78
64       789993429      789993508       1        78
65       789993508      789993586       1        77
66       789993586      789993665       2        77
67       789993665      789993745       2        78
68       789993745      789993823       1        77
69       789993823      789993902       1        78
70       789993902      789993981       1        78
71       789993982      789994061       1        78
72       789994061      789994140       1        78
73       789994140      789994219       1        78
74       789994219      789994298       1        78
75       789994298      789994377       2        77
76       789994377      789994455       1        77
77       789994455      789994535       2        78
78       789994535      789994614       0        79
79       789994614      789994692       1        77
80       789994692      789994771       1        78
81       789994771      789994850       1        78
82       789994850      789994928       1        77
83       789994928      789995007       2        77
84       789995007      789995086       1        78
85       789995086      789995165       1        78
86       789995165      789995244       1        78
87       789995244      789995322       1        77
88       789995322      789995401       2        77
89       789995401      789995480       1        78
90       789995480      789995559       1        78
91       789995559      789995638       1        78
92       789995638      789995717       1        78
93       789995717      789995796       1        78
94       789995796      789995875       1        78
95       789995875      789995953       1        77
96       789995953      789996032       1        78
97       789996032      789996111       1        78
98       789996111      789996189       1        77
99       789996189      789996268       1        78
100      789996268      789996347       1        78
101      789996347      789996426       1        78
102      789996426      789996505       1        78
103      789996505      789996583       1        77
104      789996583      789996663       2        78
105      789996663      789996742       1        78
106      789996747      789996825       1        77
107      789996826      789996904       1        77
108      789996904      789996983       2        77
109      789996983      789997062       1        78
110      789997062      789997142       2        78
111      789997142      789997220       1        77
112      789997220      789997299       2        77
113      789997299      789997378       2        77
114      789997378      789997457       2        77
115      789997457      789997536       1        78
116      789997536      789997615       1        78
117      789997615      789997694       2        77
118      789997694      789997773       1        78
119      789997773      789997852       1        78
120      789997852      789997931       1        78
121      789997931      789998010       1        78
122      789998014      789998093       1        78
123      789998093      789998172       1        78
124      789998172      789998204       1        31
```

7.6.7 Results for Two DPSlaves without Security

```
Start Time:Fri Jan 13 13:56:50 1995
Trying to factor ((10^37)+1)
Received result from RPLenstra #121
Factors of 10000000000000000000000000000000000001 = (10^37)+1
found with curve #124
Limit1 = 9037 and Limit2 = 361480
prime factors 422650073734453
prime factor (num = num/t) 2965573473134462 99
Start Time: Fri Jan 13 13:56:50 1995
Finish Time: Fri Jan 13 15:17:46 1995
Running Time: 80.9333 minutes.
There were 2 DPSlaves used.

curve  (Start Time)  (End Time)  (Run Setup Time)  (Wait Time)
3        790005411      790005490        1        77
4        790005412      790005491        1        0
5        790005490      790005568        1        75
6        790005492      790005570        1        0
7        790005569      790005648        1        76
8        790005570      790005650        2        0
9        790005648      790005728        2        77
10       790005650      790005732        1        0
11       790005728      790005810        4        77
12       790005732      790005811        1        0
13       790005810      790005888        1        75
14       790005811      790005890        2        0
15       790005888      790005967        2        75
16       790005890      790005969        2        0
17       790005967      790006047        2        77
18       790005969      790006048        1        0
19       790006047      790006126        1        77
20       790006048      790006128        1        1
21       790006126      790006205        1        76
22       790006128      790006207        1        0
23       790006205      790006286        2        77
24       790006207      790006288        2        0
25       790006286      790006366        2        77
26       790006288      790006367        1        0
27       790006366      790006444        1        76
28       790006367      790006446        1        0
29       790006444      790006524        2        76
30       790006446      790006530        2        0
31       790006529      790006608        1        77
32       790006530      790006609        1        0
33       790006608      790006687        1        76
34       790006609      790006689        2        0
35       790006687      790006766        2        76
36       790006689      790006768        1        1
37       790006766      790006845        1        76
38       790006768      790006847        1        0
39       790006845      790006924        2        76
40       790006847      790006926        1        0
41       790006924      790007003        2        76
42       790006926      790007005        1        0
43       790007003      790007082        2        76
44       790007005      790007084        1        1
45       790007082      790007161        1        75
46       790007085      790007163        1        1
47       790007161      790007240        1        75
48       790007163      790007242        2        1
49       790007240      790007319        1        76
50       790007242      790007321        1        0
51       790007319      790007398        2        76
52       790007321      790007400        1        1
53       790007398      790007477        1        76
54       790007400      790007479        1        0
55       790007477      790007556        2        76
56       790007479      790007558        1        1
57       790007556      790007635        1        76
58       790007558      790007637        1        1
59       790007635      790007714        1        76
60       790007637      790007715        1        0
61       790007714      790007793        1        76
62       790007716      790007799        1        0
63       790007798      790007877        1        77
64       790007799      790007879        1        0
65       790007877      790007956        2        76
66       790007879      790007957        1        0
67       790007956      790008035        1        76
68       790007957      790008037        2        0
69       790008035      790008115        2        77
70       790008037      790008116        1        0
71       790008115      790008194        1        77
72       790008116      790008196        1        1
73       790008194      790008273        1        75
74       790008196      790008276        2        2
75       790008273      790008352        1        74
76       790008276      790008356        2        2
77       790008353      790008432        1        75
78       790008356      790008435        1        2
79       790008432      790008511        1        75
80       790008435      790008514        1        1
81       790008511      790008590        2        75
82       790008514      790008593        1        1
83       790008590      790008670        2        76
84       790008593      790008672        1        0
85       790008670      790008751        2        78
86       790008672      790008753        1        0
87       790008751      790008831        2        77
88       790008753      790008833        1        1
89       790008831      790008910        1        76
90       790008833      790008912        1        1
91       790008910      790008989        1        76
92       790008912      790008994        1        0
93       790008993      790009072        1        77
94       790008994      790009074        1        0
95       790009072      790009152        2        77
96       790009074      790009154        1        0
97       790009152      790009231        2        75
98       790009154      790009233        2        0
99       790009231      790009311        2        76
100      790009233      790009313        2        0
101      790009311      790009391        2        77
102      790009313      790009392        1        0
103      790009391      790009470        1        76
104      790009393      790009472        1        1
105      790009470      790009549        1        76
106      790009472      790009550        1        0
107      790009549      790009628        1        76
108      790009550      790009630        2        1
109      790009628      790009707        1        76
110      790009630      790009709        1        0
111      790009707      790009786        2        76
112      790009709      790009791        1        1
113      790009786      790009868        4        76
114      790009791      790009870        1        0
115      790009868      790009948        2        77
116      790009870      790009949        1        0
117      790009948      790010027        1        77
118      790009949      790010028        1        0
119      790010027      790010105        1        76
120      790010028      790010107        1        0
121      790010106      790010185        1        77
122      790010107      790010187        1        1
123      790010185      790010264        1        76
124      790010187      790010265        1        0
```

7.6.8 Results for Four DPSlaves without Security

```
Start Time:Wed Dec  7 18:47:53 1994
Trying to factor ((10^37)+1)
Factors of 10000000000000000000000000000000000001 = (10^37)+1
found with curve #124
Limit1 = 9037 and Limit2 = 361480
prime factors 42265007373445З
prime factor (num = num/t) 296557347313446299
Start Time: Wed Dec  7 18:47:53 1994
Finish Time: Wed Dec  7 19:30:23 1994
Running Time: 42.5 minutes.
There were 4 DPSlaves used.

curve  (Start Time)  (End Time)  (Run Setup Time)  (Wait Time)
3        786847674       786847759      6       59
4        786847680       786847767      8       1
5        786847688       786847774      6       0
6        786847694       786847782      6       1
7        786847759       786847845      7       58
8        786847767       786847852      7       3
9        786847774       786847860      7       0
10       786847782       786847866      5       0
11       786847845       786847927      4       56
12       786847852       786847938      8       8
13       786847860       786847942      6       1
14       786847867       786847949      4       5
15       786847927       786848008      3       56
16       786847938       786848020      3       9
17       786847942       786848023      2       0
18       786847949       786848031      3       6
19       786848008       786848090      3       56
20       786848020       786848101      3       8
21       786848023       786848104      2       0
22       786848031       786848112      3       5
23       786848090       786848171      3       55
24       786848101       786848182      3       7
25       786848104       786848185      3       0
26       786848112       786848194      4       7
27       786848171       786848253      4       56
28       786848182       786848263      3       7
29       786848185       786848267      2       0
30       786848194       786848276      3       7
31       786848253       786848334      3       55
32       786848263       786848345      4       8
33       786848267       786848349      2       1
34       786848276       786848357      3       6
35       786848334       786848416      3       55
36       786848345       786848427      3       7
37       786848349       786848430      2       0
38       786848357       786848439      4       7
39       786848416       786848498      4       55
40       786848427       786848508      3       7
41       786848430       786848511      2       0
42       786848439       786848521      4       7
43       786848498       786848579      3       55
44       786848508       786848589      3       7
45       786848511       786848593      3       1
46       786848521       786848603      3       8
47       786848579       786848661      3       55
48       786848589       786848670      3       6
49       786848593       786848674      2       1
50       786848603       786848685      3       9
51       786848661       786848743      3       55
52       786848670       786848751      3       5
53       786848674       786848755      2       0
54       786848685       786848766      3       9
55       786848743       786848824      3       54
56       786848751       786848832      4       4
57       786848755       786848836      2       1
58       786848766       786848848      4       10
59       786848824       786848907      4       56
60       786848832       786848914      3       4
61       786848836       786848918      2       1
62       786848848       786848930      3       10
63       786848907       786848988      3       56
64       786848914       786848996      3       5
65       786848918       786849000      2       1
66       786848930       786849011      2       8
67       786848988       786849070      3       56
68       786848996       786849077      3       3
69       786849000       786849083      3       1
70       786849011       786849093      3       6
71       786849070       786849152      4       56
72       786849077       786849160      5       4
73       786849083       786849164      4       0
74       786849093       786849175      3       8
75       786849152       786849235      4       57
76       786849160       786849242      4       4
77       786849164       786849246      3       0
78       786849175       786849256      3       7
79       786849235       786849316      3       56
80       786849242       786849324      4       5
81       786849246       786849328      3       1
82       786849256       786849338      4       8
83       786849316       786849398      3       56
84       786849324       786849405      3       3
85       786849328       786849409      2       0
86       786849338       786849420      4       8
87       786849398       786849481      4       57
88       786849405       786849487      4       3
89       786849409       786849491      3       1
90       786849420       786849502      4       9
91       786849481       786849562      3       56
92       786849487       786849569      3       4
93       786849491       786849572      2       0
94       786849502       786849584      4       9
95       786849562       786849643      3       56
96       786849569       786849651      3       5
97       786849572       786849654      3       0
98       786849584       786849665      3       8
99       786849643       786849725      3       56
100      786849651       786849732      3       4
101      786849654       786849736      3       0
102      786849665       786849747      4       8
103      786849725       786849809      3       58
104      786849732       786849813      4       1
105      786849736       786849817      3       0
106      786849747       786849829      4       10
107      786849809       786849890      3       57
108      786849813       786849895      4       1
109      786849817       786849900      2       0
110      786849829       786849911      4       9
111      786849890       786849972      4       58
112      786849895       786849978      5       3
113      786849900       786849981      2       0
114      786849911       786849993      3       10
115      786849972       786850054      3       58
116      786849978       786850059      3       2
117      786849981       786850062      2       0
118      786849993       786850075      3       9
119      786850054       786850136      3       58
120      786850059       786850140      3       1
121      786850062       786850146      4       3
122      786850075       786850157      3       8
123      786850136       786850217      3       57
124      786850140       786850221      3       0
```

7.7 Results from the Factorisation of 10^59 + 1

7.7.1 Results for Seventeen DPSlaves with Security

```
Start Time: Sun Jan 15 01:07:50 1995
Trying to factor ((10^59)+1)
Factors of 10000000000000000000000000000000000000000000000000000000001 = (10^59)+1
found with curve #253
Limit1 = 36929 and Limit2 = 1477160
prime factors 1090805842068098677837
prime factor (num = num/t) 441192277099607410964453536285108 7
Start Time: Sun Jan 15 01:07:50 1995
Finish Time: Sun Jan 15 06:04:37 1995
Running Time: 296.783 minutes.
There were 17 DPSlaves used.
```

```
curve  (Start Time)  (End Time)  (Run Setup Time)  (Wait Time)
3      790132070     790132496     2      407
4      790132072     790132498     1      0
5      790132073     790132516     1      17
6      790132074     790132519     1      1
7      790132075     790132520     1      0
8      790132076     790132521     1      0
9      790132077     790132522     1      0
10     790132078     790132524     2      1
11     790132080     790132525     1      0
12     790132081     790132527     1      0
13     790132082     790133272     1      744
14     790132083     790133273     1      0
15     790132084     790133275     1      1
16     790132085     790133276     1      0
17     790132086     790133277     1      0
18     790132087     790133279     1      0
19     790132088     790133280     1      0
20     790132496     790133281     2      0
21     790132498     790133283     1      0
22     790132516     790133284     2      0
23     790132519     790133285     1      0
24     790132520     790133286     1      0
25     790132521     790133288     1      0
26     790132522     790133290     1      0
27     790132524     790133291     1      0
28     790132525     790133292     2      0
29     790132527     790133294     1      1
30     790133272     790134458     1      1163
31     790133273     790134464     1      1
32     790133275     790134465     1      1
33     790133276     790134466     1      0
34     790133277     790134467     2      0
35     790133279     790134468     1      0
36     790133280     790134470     1      1
37     790133281     790134471     2      0
38     790133283     790134472     1      0
39     790133284     790134473     1      0
40     790133285     790134475     1      1
41     790133287     790134476     1      0
42     790133288     790134477     2      0
43     790133290     790134479     1      1
44     790133291     790134480     1      0
45     790133292     790134481     1      0
46     790133294     790134482     1      0
47     790134462     790135649     1      1166
48     790134464     790135654     0      0
49     790134465     790135655     1      0
50     790134466     790135657     1      1
51     790134467     790135658     1      0
52     790134468     790135659     1      0
53     790134470     790135660     1      0
54     790134471     790135662     1      0
55     790134472     790135664     1      1
56     790134473     790135665     1      0
57     790134475     790135666     1      0
58     790134476     790135668     1      0
59     790134477     790135669     1      0
60     790134479     790135670     1      0
61     790134480     790135672     1      0
62     790134481     790135673     1      0
63     790134482     790135675     1      1
64     790135653     790136844     1      1168
65     790135654     790136849     1      0
66     790135655     790136851     1      1
67     790135657     790136852     1      0
```

```
68     790135658     790136853     1      0
69     790135659     790136854     1      0
70     790135660     790136856     2      0
71     790135662     790136857     1      0
72     790135664     790136859     1      0
73     790135665     790136860     1      0
74     790135666     790136861     2      0
75     790135668     790136863     1      0
76     790135669     790136865     1      0
77     790135670     790136867     2      1
78     790135672     790136868     1      0
79     790135673     790136869     1      0
80     790135675     790136870     1      0
81     790136848     790138027     1      1156
82     790136849     790138028     1      0
83     790136851     790138030     1      1
84     790136852     790138031     1      0
85     790136853     790138033     1      0
86     790136855     790138034     1      0
87     790136856     790138036     1      0
88     790136857     790138037     2      0
89     790136859     790138039     1      0
90     790136860     790138040     1      0
91     790136861     790138042     2      1
92     790136864     790138043     1      0
93     790136865     790138044     1      0
94     790136867     790138045     1      0
95     790136868     790138047     1      1
96     790136869     790138048     1      0
97     790136870     790138050     1      1
98     790138027     790139213     1      1162
99     790138028     790139214     1      0
100    790138030     790139216     1      0
101    790138031     790139217     2      0
102    790138033     790139219     1      0
103    790138035     790139222     1      1
104    790138036     790139223     1      0
105    790138038     790139224     1      0
106    790138039     790139226     1      1
107    790138040     790139228     1      1
108    790138042     790139229     1      0
109    790138043     790139231     1      1
110    790138044     790139232     1      0
111    790138046     790139233     0      0
112    790138047     790139235     1      1
113    790138048     790139236     1      0
114    790138050     790139238     1      1
115    790139213     790140390     1      1151
116    790139214     790140396     2      0
117    790139216     790140398     1      0
118    790139217     790140400     2      1
119    790139219     790140401     2      0
120    790139222     790140403     1      0
121    790139223     790140405     1      1
122    790139224     790140406     1      0
123    790139226     790140408     1      0
124    790139228     790140409     1      0
125    790139229     790140411     1      0
126    790139231     790140413     1      0
127    790139232     790140415     1      1
128    790139233     790140417     1      0
129    790139235     790140418     1      0
130    790139236     790140420     1      0
131    790139238     790140422     1      1
132    790140394     790141581     2      1158
133    790140396     790141587     2      0
```

134	790140398	790141588	1	0
135	790140400	790141590	1	0
136	790140401	790141592	2	1
137	790140403	790141593	1	0
138	790140405	790141595	1	0
139	790140406	790141597	2	0
140	790140408	790141598	1	0
141	790140409	790141600	2	0
142	790140411	790141602	2	1
143	790140413	790141603	1	0
144	790140415	790141605	2	1
145	790140417	790141606	1	0
146	790140418	790141608	2	0
147	790140420	790141609	1	0
148	790140422	790141611	1	0
149	790141585	790142768	2	1155
150	790141587	790142770	1	1
151	790141588	790142771	2	0
152	790141590	790142773	1	1
153	790141592	790142774	1	0
154	790141593	790142775	2	0
155	790141595	790142776	2	0
156	790141597	790142778	1	0
157	790141598	790142779	2	0
158	790141600	790142780	1	0
159	790141602	790142782	1	0
160	790141603	790142783	1	0
161	790141605	790142785	1	1
162	790141606	790142786	2	0
163	790141608	790142787	1	0
164	790141609	790142789	2	1
165	790141611	790142790	2	0
166	790142768	790143946	1	1154
167	790142770	790143947	1	0
168	790142771	790143949	1	1
169	790142773	790143950	1	0
170	790142774	790143951	1	0
171	790142775	790143953	1	0
172	790142776	790143954	2	0
173	790142778	790143956	1	1
174	790142779	790143957	1	0
175	790142780	790143958	2	0
176	790142782	790143959	1	0
177	790142783	790143960	1	0
178	790142785	790143962	1	0
179	790142786	790143963	1	0
180	790142787	790143965	1	0
181	790142789	790143967	1	0
182	790142790	790143968	2	0
183	790143946	790145135	1	1166
184	790143947	790145141	1	0
185	790143949	790145143	1	0
186	790143950	790145144	1	0
187	790143951	790145146	2	0
188	790143953	790145147	1	0
189	790143954	790145148	1	0
190	790143956	790145150	1	1
191	790143957	790145151	1	0
192	790143958	790145153	1	1
193	790143959	790145154	1	0
194	790143960	790145155	2	0
195	790143962	790145157	1	1
196	790143963	790145158	2	0
197	790143965	790145159	2	0
198	790143967	790145161	1	0
199	790143968	790145162	1	0
200	790145139	790146319	2	1156
201	790145141	790146321	2	1
202	790145143	790146326	1	0
203	790145144	790146327	2	0
204	790145146	790146329	1	0
205	790145147	790146330	1	0
206	790145148	790146331	1	0
207	790145150	790146333	1	0
208	790145151	790146334	1	0
209	790145153	790146336	1	0
210	790145154	790146337	1	0
211	790145155	790146339	1	0
212	790145157	790146340	1	0
213	790145158	790146341	1	0
214	790145159	790146343	2	1
215	790145161	790146344	1	0
216	790145162	790146345	1	0
217	790146319	790147502	1	1155
218	790146325	790147503	1	0
219	790146326	790147504	1	0
220	790146327	790147505	2	0
221	790146329	790147507	1	0
222	790146330	790147509	1	1
223	790146331	790147510	2	0
224	790146333	790147511	1	0
225	790146334	790147512	2	0
226	790146336	790147514	1	0
227	790146337	790147515	2	0
228	790146339	790147517	1	1
229	790146340	790147518	1	0
230	790146341	790147520	1	1
231	790146343	790147521	1	0
232	790146344	790147527	1	1
233	790146345	790147528	2	0
234	790147502	790148685	1	1156
235	790147503	790148687	1	1
236	790147504	790148688	1	0
237	790147505	790148691	2	0
238	790147507	790148693	1	1
239	790147509	790148694	1	0
240	790147510	790148695	1	0
241	790147511	790148697	1	1
242	790147513	790148698	1	0
243	790147514	790148700	1	0
244	790147515	790148702	1	1
245	790147517	790148703	1	1
246	790147518	790148704	1	0
247	790147520	790148705	1	0
248	790147525	790148707	1	1
249	790147527	790148708	1	0
250	790147528	790148709	1	0
251	790148685	790149873	1	1163
252	790148687	790149875	1	0
253	790148688	790149877	3	1

7.8 References

[Koy91] Yoichi Koyama, http://alpha.acast.nova.edu/simtel/ubasic.html.

[Sco89] Scott, Michael, "Factoring Large Integers on Small Computers," Working Paper: CA-0169, School of Computer Applications, Dublin City University, Dublin, Ireland.

[Sco95] Scott, Michael, The MIRACL Programming Library, available from: ftp://ftp.compapp.dcu.ie/pub/crypto/.

8. Source Code for the DistProc System

8.1 DistProc Client

```
/*
    clnhndls.C
    Brian Cox
    9th September 1993

    Distributed Processing Client - message handling routines.
*/


#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <iostream.h>
#include <sys/types.h>              /* getlogin() stuff... */
#include <unistd.h>
#include <limits.h>
#include <sys/param.h>              /* getcwd(), MAXPATHLEN */
#include "distproc.h"               /* system defines */
#include "tcp.h"                    /* network communications */
#include "messages.h"               /* message objects */
#include "clnhndls.h"


/*-----------------------------------*/
/* DistProc Client global variables. */
/*-----------------------------------*/
char * DPClient::ClientVer = "0.95";    /* client version number.
*/


/*-------------*/
/* Constructor */
/*-------------*/
DPClient::DPClient(char * srvname)
{
    int rc;

    if (srvname != NULL)
        {
        ClientID = 0;

        if (ConnectToServer(srvname) == GOOD)
            registered = GOOD;
        else
            {
            registered = FAILED;
            syserr("Could not connect to DistProc Server");
            }
        }
    else
        syserr("No DistProc Server host name given");
}


/*------------*/
/* Destructor */
/*------------*/
DPClient::~DPClient()
{
    DisconnectFromServer();

    if (DPServerHostName != NULL)
        delete [] DPServerHostName;

    cout << endl;
}


/*-------------------------------*/
/* handle a RUN_PROCESS request. */
/*-------------------------------*/
unsigned long DPClient::run(char *filename, Buffer &inbuf, Buffer
*resbuf)
{
    RunProcessMsg msg, rmsg;
    Buffer request, response;
    char pathname[MAXPATHLEN];
    int rc;

    debug("DPClient::run()");

    if (registered == FAILED)
        return FAILED;

    // debug("run: Filling in message...");

    getcwd(pathname, MAXPATHLEN);
    strcat(pathname, "/");
    strcat(pathname, filename);

    /*
    ** Fill in a RunProcessMsg message
    */
    msg.setSystemID(ClientID);
    msg.pinfo.setClientName(getlogin());
    msg.pinfo.setFileName(pathname);
    msg.pinfo.setInput(inbuf.data, inbuf.length);
    msg.pinfo.setOutputPtr(resbuf);

    /*
    ** make the request to the server.
    */
    // debug("run: Saving the msg and making the request...");
    msg.save(request);
    if (MakeRequest(request, response) == FAILED)
        {
        syswarn("couldn't send Run message");
        return INVALID_DPID;
        }

    // debug("run: reloading message");
    rmsg.load(response);

    if (rmsg.retcode == FAILED)
        {
        syswarn("Invalid pid returned from DP Server");
        return INVALID_DPID;
        }        /* 0 is an invalid pid => error */

    /*
    ** return the PID for the spawned process
    */
    // debug("run: returning retcode");
    return rmsg.pinfo.getPID();
}


/*--------------------------------*/
/* handle a KILL_PROCESS request. */
/*--------------------------------*/
int DPClient::kill(unsigned long pid)
{
    KillProcessMsg msg, rmsg;
    Buffer request, response;

    debug("DPClient::kill()");

    if ((registered == FAILED) || (pid == INVALID_DPID))
        return FAILED;

    msg.setSystemID(ClientID);
    /*
    ** fill in KILL_PROCESS message.
    */
    msg.pinfo.setPID(pid);

    /*
    ** make the request to the server.
    */
    msg.save(request);
    if (MakeRequest(request, response) == FAILED)
        {
        syswarn("couldn't send Kill message");
        return FAILED;
        }

    /*
    ** check the response to make sure that everything is ok.
    */
    rmsg.load(response);

    return rmsg.retcode;
}


/*---------------------------------------------*/
/* handle a WAIT_PROCESS request.              */
/*                                             */
/* the result is obtained from the server here! */
/*---------------------------------------------*/
int DPClient::wait(unsigned long pid)
{
    WaitProcessMsg msg, rmsg;
    Buffer request, reply, *resbuf;
    void * output;
    int outsize;

    debug("DPClient::wait()");

    if ((registered == FAILED) || (pid == INVALID_DPID))
        return FAILED;

    msg.setSystemID(ClientID);
    msg.pinfo.setPID(pid);

    /*
    ** make the request to the server.
    */
    // debug("wait: saving and sending message");
    msg.save(request);
    if (MakeRequest(request, reply) == FAILED)
        {
        syswarn("couldn't send Wait message");
        return FAILED;
        }

    /*
    ** check the response to make sure that everything is ok.
    */
    // debug("wait: reloading message");
    rmsg.load(reply);

    if (rmsg.retcode == FAILED)
        {
        syswarn("WaitProcess reply message is bad");
        return FAILED;
        }
    if (rmsg.pinfo.getStatus() == PROCESS_ABORTED)
        {
        syswarn("RemProc was abnormally terminated/aborted.");
        return FAILED;
        }

    //
    // ok, so now we need to copy the result from the process
    // that just finished to the location that was specified
    // by the Output parameter to the run_process() function.
    //

    // this gets the address of the output pointer
    // debug("wait: getting address of the output pointer");
    rmsg.pinfo.getOutputPtr((Buffer *)&resbuf);
```

```
     if (resbuf == NULL)
          {
          syserr("Address of Result Buffer is NULL");
          return FAILED;
          }
     else
          {
          //
          // make sure the output buffer is empty and
          // then copy the result into resbuf
          //
          resbuf->empty();
          rmsg.rinfo.getResult(resbuf);
          }

     // debug('wait: returning GOOD");
     return GOOD;
}


/*-------------------------------------*/
/* check if connection was successful. */
/*-------------------------------------*/
int DPClient::Registered ()
{
     return registered;
}


/*============================*/
/* Protected member functions. */
/*============================*/


/*-------------------------------------------------------------*/
/* open a connection with the server, send the request message */
/* and then read the response from the server.  Finally close  */
/* the connection with the server.                             */
/*-------------------------------------------------------------*/
int DPClient::MakeRequest(Buffer request, Buffer &response)
{
     debug("MakeRequest()");

     /*
     ** perform a request/response transaction, i.e. send the
request
     ** to the server and then wait to read take the response back
     ** from the server.
     */
     if (TransactPipe(sock,
                        request.data,
                        request.length,
                        (void **)&response.data,
                        (unsigned long *)&response.length, 0) !=
GOOD)
          {
          syswarn("couldn't send message to DPServer");
          return FAILED;
          }

     debug("MakeRequest() succeeded");

     return GOOD;
}


/*-----------------------------------------------------------*/
/* connect to the server to announce our engagement.        */
/*                                                          */
/* The following aren't used at the moment!                 */
/*                                                          */
/*  1.   what must the client send to the server?            */
/*       requester type - client                            */
/*       client address. the server can get this itself!    */
/*                                                          */
/*  2.   what must the server send back to the client?       */
/*       client id                                          */
/*                                                          */
/* Note:                                                    */
/*   I'm not sure about using a well known client socket    */
/* for special client/server system information messages.   */
/* A better way might be to let the client fork a function  */
/* which sends a SYSTEM_MSGS message to the server which    */
/* forks another function to handle any critical system     */
/* messages which may occur - i.e. Server shutting down, etc */
/*-----------------------------------------------------------*/
int DPClient::ConnectToServer(char *dpsrvhostname)
{
     ClientConnectMsg msg, rmsg;
     Buffer request, response;
     char myhostname[MAXHOSTNAMELEN];

     debug("ConnectToServer()");

     if (dpsrvhostname == NULL)
          {
          syswarn('no hostname specified for DistProc Server");
          return FAILED;
          }

     DPServerHostName = new char[strlen(dpsrvhostname)+1];
     strcpy(DPServerHostName, dpsrvhostname);

     /*
     ** set up the client information block.
     */
     gethostname(myhostname, MAXHOSTNAMELEN);
     msg.cinfo.setClientName(getlogin());
     msg.cinfo.setHostName(myhostname);
     msg.cinfo.setPort(CLIENT_TCP_PORT);

     /*
     ** open a connection with the server
     */
     if (OpenPipe(&sock, SERVER_TCP_PORT, DPServerHostName) !=
GOOD)
          {
          syswarn("opening pipe with server");
          return(FAILED);
          }

     /*
     ** Inform the server of our presence and that we are a
client.
     ** make the request to the server.
     */
     msg.save(request);
     if (MakeRequest(request, response) == FAILED)
          {
          syswarn("couldn't send Connect message");
          return FAILED;
          }

     rmsg.load(response);
     if ( rmsg.retcode == FAILED )
          {
          syswarn('DistProc Server wouldn't accept clients
connection request");
          return(FAILED);
          }
     ClientID = rmsg.cinfo.getCID();

     /*
     ** Now create a well known client socket to which the server
     ** can send system information.  See note above.
     **
     **   MakePipe(CLIENT_TCP_PORT)
     **   fork a function to handle this socket.
     */

     return rmsg.retcode;
}


/*-----------------------------------------------------------*/
/* disconnect from the server.  This will kill off any         */
/* child processes that the server is currently executing */
/* on behalf of the client;                                  */
/*-----------------------------------------------------------*/
int DPClient::DisconnectFromServer()
{
     ClientDisconnectMsg msg, rmsg;
     Buffer request, response;

     debug('DPClient::DisconnectFromServer()");

     if (registered == FAILED)
          return FAILED;

     msg.setSystemID(ClientID);
     /*
     ** tell the server that we're closing down.
     ** make the request to the server.
     */
     // debug("DisconnectFromServer: saving message and making
request");
     msg.save(request);
     if (MakeRequest(request, response) == FAILED)
          {
          syswarn("couldn't send Disconnect message");
          return FAILED;
          }

     // debug("DisconnectFromServer: reloading reply");
     rmsg.load(response);

     if (ClosePipe(sock) != GOOD)
          {
          syswarn("closing the pipe");
          return(FAILED);
          }

     // debug('DisconnectFromServer: pipe closed and returning
retcode");
     return rmsg.retcode;
}
```

```
/*
     rpargio.C
     Brian Cox

     Remote Process ARGument Input/Output.
     These functions take care of receiving input arguments and
     returning the results to the Slave first and finally back
     to the Client Application via the Server.
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <iostream.h>
#include "distproc.h"
#include "messages.h"
#include "tcp.h"

//
// Global variables.
//
char            *HostNameForSlave;
char            *DPServerHostName;    // name of DistProc Server's
host
unsigned long  RemProcID;
unsigned long  SlaveID;


/*-----------------------------------------------------------------*/
/* open a connection with the server, send the request message */
/* and then read the response from the server.  Finally close  */
/* the connection with the server.                              */
/*-----------------------------------------------------------------*/
int MakeRequest(Buffer request, Buffer &response)
{
     int sock;

     debug("RemProc MakeRequest()");

     //
     // open a connection with the server
     //
     if (OpenPipe(&sock, SERVER_TCP_PORT, DPServerHostName) !=
GOOD )
          {
          syswarn('opening pipe to Server");
          return(FAILED);
          }

     //
     // perform a request/response transaction, i.e. send the
request
     // to the server and then wait to read take the response back
     // from the server.
     //
     if ( TransactPipe(sock, request.data, request.length,
                         (void **)&response.data,
                         (unsigned long *)&response.length, 60) !=
GOOD )
          syswarn("in pipe transaction");

     if ( ClosePipe(sock) != GOOD )
          {
          syswarn("closing pipe");
          return(FAILED);
          }

     return(GOOD);
}


int GetInputParameters (Buffer &inbuf)
{
     Buffer rpidbuf, sidbuf; // buffer to hold slave id and
remproc id.
     int sock;
     char myhostname[MAXHOSTNAMELEN];
     unsigned long len;
     int rc;

     debug("GetInputParameters()");

     /*
     ** set up the client information block.
     */
     gethostname(myhostname, MAXHOSTNAMELEN);
     HostNameForSlave = (char *) malloc(strlen(myhostname)+1);
     strcpy(HostNameForSlave, myhostname);

     if (OpenPipe(&sock, SLAVE_REMPROC_TCP_PORT, HostNameForSlave)
== FAILED)
          {
          syswarn("couldn't open Slaves RemProc socket to get
process arguments.");
          return FAILED;
          }

     /*
     ** read in:   the system id for this remproc,
     **            the slaves id number,
     **            the distproc servers host name,
     **            the input parameters.
     */
     ReadPipe(sock, (void **)&rpidbuf.data, (unsigned long
*)&rpidbuf.length, 120);
     memcpy(&RemProcID, rpidbuf.data, rpidbuf.length);

     ReadPipe(sock, (void **)&sidbuf.data, (unsigned long
*)&sidbuf.length, 120);
     memcpy(&SlaveID, sidbuf.data, sidbuf.length);

     ReadPipe(sock,  (void **)&DPServerHostName, &len, 120);

     ReadPipe(sock, (void **)&inbuf.data, (unsigned long
*)&inbuf.length, 12);


     ClosePipe(sock);

     return GOOD;
}


int ReturnResults (Buffer &resbuf)
{
     ResultMsg msg, rmsg;
     Buffer result, reply;
     ResultInfo rinfo;
     int sock;

     debug('ReturnResults()");

     msg.setSystemID(SlaveID);
     msg.rinfo.setPID(RemProcID);
     msg.rinfo.setResult(resbuf.data, resbuf.length);
     msg.retcode = GOOD;     // change this to reflect the RemProc
retcode
     msg.save(result);

     if (MakeRequest(result, reply) == FAILED)
          {
          syswarn("couldn't send ReturnResult message to Server");
          return FAILED;
          }

     rmsg.load(reply);
     if ( rmsg.retcode != GOOD )
          {
          syswarn("returning result to DistProc Server");
          return FAILED;
          }

     return GOOD;
}
```

8.2 DistProc Server

```
/*
;       dpserver.C
;       Brian Cox
;       9th September 1993
;
;       Distributed Processing Server.
*/

#define debug_on

#include <iostream.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <signal.h>
#include "distproc.h"
#include "tcp.h"
#include "messages.h"
#include "srvhndls.h"

// DistProc Server object!
DPServer *dpsrv = (DPServer *)NULL;

/*
; Function to handle incoming requests.
*/
void HandleRequests()
{
     int sock, rc;

     debug("HandleRequests()");

     do
          {
          if ((rc = dpsrv->WaitRequest(sock)) > 0)
               rc = dpsrv->DispatchRequest(sock);
          }
     while (dpsrv->ServerOK == TRUE);
}

/*
; All requests to the DistProc Server from various Clients and
; Slaves must be aborted first.  The abort message should show
that
; the Server is shutting down.  A signal could be sent to each
; process on the server which is currrently engaged to a
Client/Slave
; and the appropriate message could be sent.
*/
void ShutDown(int status)
{
     if (dpsrv == NULL)
          {
          cout << "\nDPServer interrupted. Closing down.\n" <<
flush;
          exit(1);
          }
     else
          dpsrv->ServerOK = FALSE;
//              dpsrv->DPServer::~DPServer();
}

/*
; main server function.
*/
int main(int argc, char *argv[])
{
     setvbuf(stderr, (char*)NULL, _IONBF, 0);
     setvbuf(stdout, (char*)NULL, _IONBF, 0);

     signal(SIGINT, ShutDown);      /* interrupt */
     signal(SIGTERM, ShutDown);     /* terminate */

     dpsrv = new DPServer;

     if (dpsrv == NULL)
          {
          cout << "\nError: Couldn't create DPServer.\n" << flush;
          return FAILED;
          }
     if (dpsrv->ServerOK == FALSE)
          {
          cout << "\nCouldn't initialise DPServer.\n" << flush;
          delete dpsrv;
          return FAILED;
          }

     cout << "\nDistProc Server v" << dpsrv->Version << "
started.\n" << flush;

     HandleRequests();

     delete dpsrv;

     cout << "\nDistProc Server v" << dpsrv->Version << " shut
down.\n" << flush;

     return;
}
```

```
/*
     srvhndls.C
     Brian Cox
     September 1993

     Distributed Processing Server - message handling routines.
*/

#define debug_on

#include "srvhndls.h"
#include "waitrm.h"
#include <sys/select.h>
#include <sys/time.h>


DPServer::DPServer()
     :DPServerPrimitives()
{
}


DPServer::~DPServer()
{
}


/*
;
; The return values from select() are as follows:
;
;    = 0 :  time out
;    > 0 :  number of ready descriptors in the sets
;     -1 :  failure, errno set, descriptors not changed.
;
*/
DPServer::WaitRequest(int &newsock)
{
     int clilen, rc, fd, ssock;
     struct sockaddr_in cli_addr;

     debug("DPServer::WaitRequest()");

     ClearUpErrors();

     while (1)
          {
          tout.tv_sec = 60;        // times out after 60 seconds.
          tout.tv_usec = 0;
          bcopy((char *)&afds, (char *)&rfds, sizeof(rfds));
          rc = select(nfds, &rfds, NULL, NULL, &tout);

          //
          // Make sure select returned OK
          //
          if (rc < 0)
               {
               syserr("select error");
               return rc;  // ----- returning -----
               }

          else
               //
               // Check if select timed out
               //
               if (rc == 0)
                    {
                    cout << "\nSelect timed out!!!\n" << flush;
                    return rc;  // ----- returning -----
                    }

          else
               //
               // Check the main server socket for activity.
               //
               if (FD_ISSET(serversock, &rfds))
                    {
                    cout << "\nActivity on server socket #" <<
serversock;
                    WaitConnectPipe(serversock, &ssock, &cli_addr,
&clilen);
                    if (ssock < 0)
                         {
                         syserr("accept failed");
                         return -1;
                         }
                    int so_flag = 1; // non zero value turns
option/flag on
                    if (setsockopt(ssock, SOL_SOCKET, SO_KEEPALIVE,
(char*) &so_flag, sizeof(so_flag)) < 0)
                         syserr("SO_KEEPALIVE setsockopt error");

                    //
                    // Add new socket to the set of active sockets,
and loop again.
                    //
                    FD_SET(ssock, &afds);
                    }

          else
               //
               // If there's activity on a socket other than the
server
               // socket break out of loop.
               //
               if (rc > 0)
                    {
                    //
                    // Check each socket in tha active set for
activity
                    //
                    for (fd=0; fd<nfds; fd++)
                         if (fd != serversock && FD_ISSET(fd,
&rfds) )
                              {
                              newsock = fd;
                              cout << "\nActivity on socket #" <<
newsock;
                              return rc;
                              }
                    if (fd == nfds)     // couldn't find active fd
                         syserr("could not find active file
descriptor");
                    }

          } // end while


     return rc;
}


/*
; For the moment this is a serialised Server. It probably
; should fork a process to handle the incoming request.
; The reasons for forking() a sub process to handle request are:
;
;    1.   multiple DistProc Requesters
;
;    2.   DistProc message from client/slave which effects either
;         the Slave process (ClientShutDown or ServerShutDown) or
;         a sub process it is executing on behalf of the DistProc
;         Server (e.g. KillProcess)
;
; A process should also be forked() for each WaitProcess request.
*/
DPServer::DispatchRequest(int sock)
{
     Service sv = no_service;
     Buffer b;
     BaseMsg msg;                  /* base message object */
     int rc;

     debug("DispatchRequest");

     /* read the request from the pipe */
     rc = ReadPipe(sock, (void **)&b.data, (unsigned long
*)&b.length, 60);
     if (rc == FAILED)
          {
          syserr("could not read request");
          cerr << "Error reading on socket #" << sock << endl;
          //-------------------------------------------------------
---
          //
          //  Use ErrorOnSock() instead of handling the error
here!!
          //
          //-------------------------------------------------------
---
          ErrorOnSocket(sock);
//             ClosePipe(sock);
//             FD_CLR(sock, &afds);
          return rc;
          }

     //
     // make sure that b is not NULL before calling msg.load()
     //
     msg.load(b);                  /* recreate the message object */
     sv = msg.getService();        /* what's the request? */

     switch (sv)
          {
          /*-----------------------------------------------------
           The following messages can only come from a Client.
          -----------------------------------------------------*/
          case client_connect   : ClientAttachHnd(sock, b);
                                  break;
          case run_proc         : RunProcessHnd(sock, b);
                                  break;
          case kill_proc        : KillRemProcHnd(sock, b);
                                  break;
          case client_shutdown  : ClientDetachHnd(sock, b);
                                  break;
          /*
          ; wait_proc is a special case.  If no result is ready
          ; then the socket cannot be closed yet!  Therefore
          ; WaitProcess() takes care of closing the socket itself.
          */
          case wait_proc        : WaitProcessHnd(sock, b);
                                  return(GOOD);

          /*-----------------------------------------------------
           The following messages can only come from a Slave.
          -----------------------------------------------------*/
          case slave_connect    : SlaveAttachHnd(sock, b);
                                  break;
          case slave_shutdown   : SlaveDetachHnd(sock, b);
                                  break;
          case result           : ResultFromRemProcHnd(sock, b);
                                  break;
          case death_of_child   : DeathOfChildHnd(sock, b);
                                  break;

          /*-------------------------------------------------------
           For the moment unknown messages are simply discarded.
          -------------------------------------------------------*/
          default : cerr << "\nDPServer: unknown message received
on socket #"
                         << sock << endl << flush;
                    break;        // add code to handle unknown
messages
          }

     return rc;
}


/*=================================================================*
/
/*    H A N D L E R S   f o r   C L I E N T   R E Q U E S T S
*/
/*=================================================================*
/

/*--------------------------------------------*/
/* receive a connection request from a client. */
/*--------------------------------------------*/
int DPServer::ClientAttachHnd(int sock, Buffer &b)
```

```
{
     ClientConnectMsg msg(b);
     Buffer response;
     int rc;

     msg.cinfo.setSocket(sock);
     rc = ClientAttach(msg);          /* this sets cinfo.cid */

     msg.retcode = rc;
     msg.save(response);
     FulfillRequest(sock, response);

     return rc;
}


/*----------------------------*/
/* a client is disconnecting. */
/*----------------------------*/
int DPServer::ClientDetachHnd(int sock, Buffer &b)
{
     ClientDisconnectMsg msg(b);
     Buffer response;
     int rc;

     rc = ClientDetach(msg);

     msg.retcode = rc;
     msg.save(response);
     FulfillRequest(sock, b);

     //
     // remove clients socket from the select descriptor set
     //
     FD_CLR(sock, &afds);
     ClosePipe(sock);

     return rc;
}


/*--------------------------------------------*/
/* Handle a RUN_PROCESS request from a client. */
/*--------------------------------------------*/
int DPServer::RunProcessHnd(int sock, Buffer &b)
{
     RunProcessMsg msg(b);
     Buffer reply;
     int rc;

     rc = RunProcess(msg);

     /*
     ; Send the response from the Slave back to the Client.
     */
     msg.save(reply);
     FulfillRequest(sock, reply);

     return rc;
}


/*--------------------------------------------------------*/
/* this is a request from a client to wait for a RemProc */
/* to terminate.                                          */
/*--------------------------------------------------------*/
int DPServer::WaitProcessHnd(int sock, Buffer &b)
{
     WaitProcessMsg msg(b);
     int rc;

     rc = WaitProcess(msg, sock);

     return rc;
}


/*-----------------------------------------------*/
/* handle a KILL_PROCESS request from the Client. */
/*-----------------------------------------------*/
int DPServer::KillRemProcHnd(int sock, Buffer &b)
{
     KillProcessMsg msg(b);
     Buffer reply;
     int rc;

     rc = KillProcess(msg.pinfo.getPID());

     msg.retcode = rc;
     msg.save(reply);
     rc = FulfillRequest(sock, reply);

     return rc;
}


/*==============================================================*/
/*    H A N D L E R S   f o r   S L A V E   R E Q U E S T S    */
/*==============================================================*/

/*----------------------------------------------*/
/* receive a registration request from a slave. */
/*----------------------------------------------*/
int DPServer::SlaveAttachHnd(int sock, Buffer &b)
{
     SlaveConnectMsg msg(b);
     Buffer response;
     int rc;

     msg.sinfo.setSocket(sock);
     rc = SlaveAttach(msg);

     msg.retcode = rc;
     msg.save(response);
     FulfillRequest(sock, response);

     return rc;
}
```

```
/*--------------------------*/
/* A Slave is closing down. */
/*--------------------------*/
int DPServer::SlaveDetachHnd(int sock, Buffer &b)
{
     SlaveDisconnectMsg msg(b);
     Buffer reply;
     int rc;

     //
     // Acknowledge Slaves message - shuts down regardless of
retcode.
     //
     msg.retcode = GOOD;
     msg.save(reply);
     FulfillRequest(sock, reply);
     //
     // remove slaves socket from the select descriptor set
     //
     ClosePipe(sock);
     FD_CLR(sock, &afds);

     rc = SlaveDetach(msg);

     return GOOD;
}


/*-----------------------------------------*/
/* Handle a RESULT from a RemProc process. */
/*-----------------------------------------*/
int DPServer::ResultFromRemProcHnd(int sock, Buffer &b)
{
     ResultMsg msg(b);
     Buffer response;
     int rc;

     //
     // send back OK to the Slave in question,
     // it's our responsibility now....
     //
     msg.retcode = GOOD;
     msg.save(response);
     FulfillRequest(sock, response);

     //
     // Remove the socket connect with the RemProc immediately!
     //
     ClosePipe(sock);
     FD_CLR(sock, &afds);

     rc = RemProcReturningResult(msg);

     return rc;
}


/*--------------------------*/
/* A Slave is closing down. */
/*--------------------------*/
int DPServer::DeathOfChildHnd(int sock, Buffer &b)
{
     DOCMsg msg(b);
     Buffer reply;
     int rc;

     rc = DeathOfChildRemProc(msg);

     /*
     ; Acknowledge Slaves message .
     */
     msg.retcode = rc;
     msg.save(reply);
     FulfillRequest(sock, reply);

     return GOOD;
}
```

```
/*
   srvprim.C
   Brian Cox
   March '94

   DPServer's distributed processing primitives member functions.
*/

#define debug_on


#include "srvprim.h"

const char * const DPServerPrimitives::Version = "0.95";


/*-------------*/
/* Constructor */
/*-------------*/
DPServerPrimitives::DPServerPrimitives()
{
     int listenqsize = 20;  // # of listens that are queued up for
us

     /*
     ** create the well known server socket to
     ** which all slaves and clients attach
     */
     if (MakePipe(&serversock, SERVER_TCP_PORT, listenqsize) != 0)
          {
          syserr("making the Servers socket");
          ServerOK = FALSE;
          }
     else
          {
          int so_flag = 1; // non zero value turns option/flag on
          if (setsockopt(serversock, SOL_SOCKET, SO_KEEPALIVE,
(char*) &so_flag, sizeof(so_flag)) < 0)
               syserr("SO_KEEPALIVE setsockopt error");

          ServerOK = TRUE;
          nfds = getdtablesize();
          FD_ZERO(&afds);
          FD_SET(serversock, &afds);
          }
}


/*------------*/
/* Destructor */
/*------------*/
DPServerPrimitives::~DPServerPrimitives()
{
     /*
          we're really just informing the clients and slaves that
          we are shutting down - it's up to each requester to
          take the appropriate action.
     */
     InformClientsOfShutdown();

     if (InformSlavesOfShutdown() == FAILED)
          syserr("couldn't inform Slaves of Server shutdown");

     /*
     ** Close the original socket.
     */
     ClosePipe(serversock);
}


/*---------------*/
/* ErrorOnSocket */
/*---------------*/
int DPServerPrimitives::ErrorOnSocket (int sock)
{
     int rc, reqtype;
     unsigned long id;
     SlaveInfo sinfo;
     ClientInfo cinfo;

     //
     // Go through the slave tables looking for the socket in
question
     //
     rc = slaverm.FindFirst(sinfo);
     while (rc == GOOD)
          if (sinfo.getSocket() == sock)
               {
               id = sinfo.getSID();
               reqtype = SLAVE_REQUESTER;
               cout << "Reading/Writing to DPSlave #" << id <<
endl << flush;
               break; // ------- break out of while loop --------
               }
          else
               rc = slaverm.FindNext(sinfo);

     //
     // if rc == FAILED then
     //    Socket is not associated with a slave, so check the
clients.
     //    This works cause we break out of the previous loop.
     //
     if (rc == FAILED)
          {
          //
          // Go through the client tables looking for the socket
in question
          //
          rc = clientrm.FindFirst(cinfo);
          while (rc == GOOD)
               if (cinfo.getSocket() == sock)
                    {
                    id = cinfo.getCID();
                    reqtype = CLIENT_REQUESTER;
                    cout << "Reading/Writing to DPClient #" << id
<< endl << flush;
                    break; // ------- break out of while loop -----
---
                    }
               else
                    rc = clientrm.FindNext(cinfo);

          //
          // If rc == FAILED then the socket is not associated
with either
          // a DPClient or a DPSlave, so it must be a RemProc
socket!
          //
          if (rc == FAILED)
               {
               ClosePipe(sock);
               FD_CLR(sock, &afds);

               syserr("error on a RemProc socket, while receiving
a result");
               //
               // Don't need to do anything as the DPSlave should
return a DOCMsg
               // and we will then see that the RemProc never
returned a result before
               // it died!
               //
               return FAILED;
               }
          }

     if (reqtype == SLAVE_REQUESTER)
          {
          if (sinfo.getStatus() == SLAVE_UNAVAILABLE)
               {
               syserr("Cannot fulfill request to Slave as it is
UNAVAILABLE/DEAD");
               return FAILED;
               }
          }
     else
          if (cinfo.getStatus() == CLIENT_UNAVAILABLE)
               {
               syserr("Cannot fulfill request to Client as it is
UNAVAILABLE/DEAD");
               return FAILED;
               }

     ErrorOnSocket(sock, id, reqtype);

     return GOOD;  // regardless!!
}


/*--------------------------------------------------------------*/
/* Set the status of the requester to UNAVAILABLE.              */
/* Log the error and relevant information in the error queue. */
/*--------------------------------------------------------------*/
void DPServerPrimitives::ErrorOnSocket(int sock, unsigned long id,
int reqtype)
{
     SlaveInfo sinfo;
     ClientInfo cinfo;
     ErrInfo err;
     int rc;

     debug("\nErrorOnSocket\n");

     err.sock = sock;
     err.id = id;
     err.reqtype = reqtype;

     if (reqtype == SLAVE_REQUESTER)
          {
          slaverm.Find(id, sinfo);
          sinfo.setStatus(REQ_UNAVAILABLE);
          slaverm.Update(sinfo);
          }
     else
          {
          clientrm.Find(id, cinfo);
          cinfo.setStatus(REQ_UNAVAILABLE);
          clientrm.Update(cinfo);
          }

     if (ErrQ.put(err) != 0)
          syserr("error queue full");
}

/*-----------------------------------------------------------------
*/
/* Clear up any outstanding errors in the error queue.
*/
/* Close the socket and delete it from the active set of sockets.
*/
/*-----------------------------------------------------------------
*/
void DPServerPrimitives::ClearUpErrors()
{
     ErrInfo err;
     ClientDisconnectMsg clnmsg;
     SlaveDisconnectMsg  slvmsg;

     if (ErrQ.getItemCount() > 0)
          {
          debug("\nClearUpErrors\n");

          while (ErrQ.get(err) == 0)
               {
               if (err.reqtype == SLAVE_REQUESTER)
                    {
                    //
                    // Error with a connection to a slave.
                    //
                    syswarn("error on a DPSlave socket");
                    slvmsg.setSystemID(err.id);
                    SlaveDetach(slvmsg);
                    }
               else if (err.reqtype == CLIENT_REQUESTER)
                    {
                    //
                    // Error with a connection to a client.
                    //
                    syawarn("error on a DPClient socket");
                    clnmsg.setSystemID(err.id);
                    ClientDetach(clnmsg);
                    }

               ClosePipe(err.sock);
               FD_CLR(err.sock, &afds);
               }
          }
}


/*-------------------------------------------------------------------
-----*/
/* send the request message and then read the response from the
slave. */
/*
*/
/* at the moment the Server doesn't make any requests to the
client,   */
/* however this might change in order to facilitate better error
*/
/* handling.
*/
/*-------------------------------------------------------------------
-----*/
int DPServerPrimitives::MakeRequest(unsigned long sid, Buffer
request, Buffer &reply)
{
     SlaveInfo sinfo;
     int sock, rc, retrycount = 10;

     debug("MakeRequest");

     slaverm.Find(sid, sinfo);
     if (sinfo.getStatus() == SLAVE_UNAVAILABLE)
          {
          syserr("Cannot contact Slave as it is
UNAVAILABLE/DEAD");
          return FAILED;
          }

     //
     // perform a request/response transaction, i.e. send the
request
     // to the server and then wait to read take the response back
     // from the server.
     //
     sock = sinfo.getSocket();

     cout << " - Making request to DPSlave #" << sid << endl <<
flush;

     if ( (rc = TransactPipe(sock, request.data, request.length,
(void **)&reply.data,
                    (unsigned long *)&reply.length, 60)) == FAILED)
          {
          syserr("MakeRequest(): transaction failed");
          ErrorOnSocket(sock, sid, SLAVE_REQUESTER);
          }

     return rc;
}


/*-----------------------------------------------------------*/
/* send back the filled in message to the requester.     */
/* The reason for the name change in the 2nd parameter */
/* is that to fulfill a request from a server          */
/* a Server must return a response - this is it.       */
/*-----------------------------------------------------------*/
int DPServerPrimitives::FulfillRequest(int sock, Buffer response)
{
     SlaveInfo sinfo;
     ClientInfo cinfo;
     unsigned long id;
     int reqtype;
     int rc;

     debug("FulfillRequest");

     cout << " - Fulfilling request to DPRequester on socket #"
          << sock << endl << flush;

     //
     //  Send the message to the requester - Client/Slave.
     //
     if (WritePipe(sock, response.data, response.length) ==
FAILED)
          {
          syserr("FulfillRequest()");
          ErrorOnSocket(sock);
          return FAILED;
          }

     return GOOD;
}


/*-------------------------------------------------*/
/* receive a connection request from a client. */
/*-------------------------------------------------*/
int DPServerPrimitives::ClientAttach (ClientConnectMsg &msg)
{
     int rc;

     debug("ClientAttach()");

     /*
     ;  Get the requesters info from the message.
     ;  Add it to the Client System table and update the
     ;  message.
     */
     rc = clientrm.Attach(msg.cinfo);                 /* this sets
cinfo.cid */

     return rc;
}

/*-------------------------------------------------------*/
```

```
/* a client is disconnecting.....                       */
/* so we must stop all it's RemProcs and clean up       */
/* the system tables...don't forget to clean out the */
/* results table also!                                  */
/*------------------------------------------------------*/
int DPServerPrimitives::ClientDetach(ClientDisconnectMsg &msg)
{
     ProcessInfo pinfo;
     int rc;

     debug("ClientDetach");

     rc = procrm.FindFirst(pinfo);
     while (rc == GOOD)
          {
          if (pinfo.getCID() == msg.getSystemID())
               KillProcess(pinfo.getPID());  // ignore return
code!
          rc = procrm.FindNext(pinfo);
          }

     rc = clientrm.Detach(msg.getSystemID());

     return rc;
}


/*----------------------*/
/* runprocess primitive */
/*----------------------*/
int DPServerPrimitives::RunProcess (RunProcessMsg &msg)
{
     Buffer request, reply;
     unsigned long sid;
     int rc, i;

     debug("RunProcess()");

     for (i=0; i<slaverm.NumSlaves; i++)
          {
          //
          // try and allocate a slave
          //
          if ((rc = slaverm.Alloc(sid)) == GOOD)
               {
               msg.pinfo.setSID(sid);
               //
               // try and allocate a new process entry
               //
               if ((rc = procrm.Alloc(msg.pinfo)) == GOOD)
                    {
                    //
                    // send runprocess msg to allocated slave
                    // if it worked, update process table and break
out of loop
                    //
                    msg.save(request);
                    if ((rc = MakeRequest(sid, request, reply)) ==
GOOD)
                         {
                         msg.load(reply);
                         if ( (rc = msg.retcode) == GOOD)
                              {
                              msg.pinfo.setStatus(PROCESS_RUNNING);
                              procrm.Update(msg.pinfo);

                              break;  // ----- break out of for
loop -----

                              }
                         }
                    //
                    // rollback process allocation
                    //
                    debug("RunProcess: undoing process
allocation");
                    procrm.Dealloc(msg.pinfo.getPID());
                    }

               //
               // rollback the Slave allocation
               //
               debug("RunProcess: undoing slave allocation");
               slaverm.Dealloc(msg.pinfo.getSID());
               }
          else
               {
               syserr("error in allocating a DPSlave");
               break;  // ----- break out of for loop -----
               }
          }

     msg.retcode = rc;
     return rc;
}


/*------------------------*/
/* kill process primitive */
/*------------------------*/
int DPServerPrimitives::KillProcess (unsigned long pid)
{
     KillProcessMsg msg;
     Buffer request, reply;
     int rc;

     debug("KillProcess()");

     //
     // Get the actual pid of the process on the DistProc Slave.
     // Delete the pid from the Process system table and
     // send a KILL_PROCESS to the appropriate Slave.
     //

     // find the process entry
     if ((rc = procrm.Find(pid, msg.pinfo)) == GOOD)
          {
          // delete the process entry
          procrm.Dealloc(pid);
          if (msg.pinfo.getStatus() == PROCESS_RUNNING)
               {
                // send the kill message
                msg.save(request);
                rc = MakeRequest(msg.pinfo.getSID(), request,
reply);
                }
            else
                {
                // delete the result
                resrm.Delete(pid);
                msg.retcode = GOOD;
                msg.save(reply);
                }
            }

     return rc;
}


/*---------------------------------------------*/
/* receive a registration request from a slave. */
/*---------------------------------------------*/
int DPServerPrimitives::SlaveAttach (SlaveConnectMsg &msg)
{
     int rc;

     debug("SlaveAttach()");

     rc = slaverm.Attach(msg.sinfo);

     return rc;
}

/*--------------------------*/
/* A Slave is closing down. */
/*--------------------------*/
int DPServerPrimitives::SlaveDetach (SlaveDisconnectMsg &msg)
{
     ResultMsg dummyresultmsg;
     unsigned long sid, pid;
     int rc, i;
     ProcessInfo pinfo;

     debug("SlaveDetach()");
     sid = msg.getSystemID();

     //
     // remove the Slave in question from the Slave system table
     // before trying to reallocate its' processes.
     //
     slaverm.Detach(sid);

     //
     // for each process in the system table
     //    check if it is executing on the Slave sid
     //    if it is then
     //      start the process running again on its new Slave
     //
     rc = procrm.FindFirst(pinfo);
     for (i = 0; i < procrm.NumProcesses; i++)
          {
          if ((pinfo.getSID() == sid) && (pinfo.getStatus() ==
PROCESS_RUNNING))
               {
               //
               // Set result messages retcode to FAILED.
               // Update the process tables to show that the
process failed.
               // Call RemProcReturningResult with dummy message
to handle results & waits
               //
               dummyresultmsg.retcode = FAILED;
               dummyresultmsg.rinfo.setPID(pinfo.getPID());
               pinfo.setStatus(PROCESS_ABORTED);
               procrm.Update(pinfo);
               RemProcReturningResult(dummyresultmsg);
               }
          rc = procrm.FindNext(pinfo);
          }

     return GOOD;
}


/*----------------------------------------------*/
/* Handle a RESULT from a RemProc process.      */
/* A WaitProcess call could be waiting on this  */
/* so that it can complete.                     */
/*                                              */
/* Don't forget to close the socket if a wait is */
/* pending on this result!                      */
/*----------------------------------------------*/
int DPServerPrimitives::RemProcReturningResult(ResultMsg &msg)
{
     WaitProcessMsg rmsg;
     Buffer resultreply;
     ProcessInfo pinfo;
     unsigned long pid;
     int clientsock;
     int rc;

     debug("RemProcReturningResult()");

     //
     // look up pid in the process system table
     //
     if (procrm.Find(msg.rinfo.getPID(), pinfo) == FAILED )
          {
          syswarn("couldn't find process info");
          return FAILED;
          }

     //
     // ok, that's the normal stuff done - now check for any
outstanding
     // Wait requests in the WaitRemProcQueue looking for our
current
     // result.
     //
     pid = pinfo.getPID();
     if (waitrm.Get(pid, clientsock) == GOOD)
          {
          rmsg.pinfo = pinfo;
          rmsg.rinfo = msg.rinfo;
          rmsg.save(resultreply);
          rc = FulfillRequest(clientsock, resultreply);
          if (rc == FAILED)
               syserr("could not return result to client");

          procrm.Dealloc(pid);     // clear process table
          }
     else
          //
          // update the process table only if the process is still
          // down as running.  Don't do anything if it's aborted
etc.
          //
          {
          if (pinfo.getStatus() == PROCESS_RUNNING)
               {
               pinfo.setStatus(PROCESS_FINISHED);
               procrm.Update(pinfo);
               }
          //
          // add the new results to the results system table.
          //
          rc = resrm.Add(msg.rinfo);
          }

     return rc;
}


/*------------------------------------------------------------*/
/* this is a request from a client to wait for a RemProc to  */
/* terminate.                                                 */
/*                                                            */
/* MAKE SURE THAT THERE ARE NOT MULTIPLE WAITS ON ONE PID!    */
/*                                                            */
/* search the results table for a suitable result             */
/* if (one is found) then                                     */
/*    copy it and then delete it from the table.              */
/*    send back to the Client.                                */
/*    clean out process table etc.                            */
/*    unlock tables.                                          */
/* else                                                       */
/*    add this request to the WaitRemProc queue.              */
/*                                                            */
/* srvResultFromSlave will look at the WaitRemProcQueue when */
/* it is called to see if there are any outstanding WaitAny  */
/* requests on its current result.                            */
/*------------------------------------------------------------*/
int DPServerPrimitives::WaitProcess (WaitProcessMsg &msg, int
sock)
{
     ProcessInfo pinfo;
     unsigned long pid;
     Buffer reply;
     int rc;

     debug("WaitProcess()");

     pid = msg.pinfo.getPID();

     //
     // Make sure that the process id # is actually in the system,
     // if not then return an error.
     //
     if (procrm.Find(pid, pinfo) == FAILED)
     {
          syswarn("couldn't find Process ID # in system tables");
          msg.retcode = FAILED;
          msg.save(reply);
          FulfillRequest(sock, reply);
          return FAILED;
     }

     //
     // Check the results table to see if the result is present.
     // In other words make sure that the process is not running
any more!
     // This way we cover the chance that it was aborted or core
dumped.
     //
     if (pinfo.getStatus() != PROCESS_RUNNING)
     {
          rc = resrm.Find(pid, msg.rinfo);
          debug("\tresult is in the Result System Table!");
          msg.pinfo = pinfo;
          msg.save(reply);
          FulfillRequest(sock, reply);

          resrm.Delete(pid);                  // clear the result
table
          procrm.Dealloc(pid);     // clear the process table
     }
     else
     {
          debug("\tresult is not ready yet - adding request to
WaitRemProc Queue.");
          //
          // add the Wait request to the WaitRemProcQueue
          // along with the socket in use.
          //
          rc = waitrm.Add(pid, sock); // pairs with GetWaitRemProc
in ResultFromSlave
     }

     debug("returning from WaitProcess()...");
     return rc;
}


/*---------------------*/
/* DeathOfChildRemProc */
/*---------------------*/
int DPServerPrimitives::DeathOfChildRemProc (DOCMsg &msg)
{
     ResultMsg dummyresultmsg;
     unsigned long pid;
     ProcessInfo pinfo;
```

```
    int finished, found = FALSE;

    debug("DeathOfChildRemProc()");

    /*
        find the process in the process table
        check it's status.
        if it's down as still working then
            mark it as ABORTED
    */
    finished = procrm.FindFirst(pinfo);
    while ((finished != FAILED) && (found == FALSE))
        {
        if ((pinfo.getSID() == msg.sid) && (pinfo.getActualPID()
== msg.realpid))
            {
            found = TRUE;
            if (pinfo.getStatus() == PROCESS_RUNNING)
                {
                //
                // update the process tables to show that the
process failed
                // set result msg's retcode to FAILED
                // finally call RemProcReturningResult to
handle results + waits
                //

                pinfo.setStatus(PROCESS_ABORTED);
                procrm.Update(pinfo);

                dummyresultmsg.retcode = GOOD;
                dummyresultmsg.rinfo.setPID(pinfo.getPID());
                RemProcReturningResult(dummyresultmsg);
                }
            }
        else
            finished = procrm.FindNext(pinfo);
        }

    return GOOD;
}


void DPServerPrimitives::InformClientsOfShutdown ()
{
    /* it's up to Clients to fend for themselves! */
}


int DPServerPrimitives::InformSlavesOfShutdown ()
{
    ServerShutDownMsg msg;
    Buffer request, reply;
    SlaveInfo sinfo;

    msg.save(request);

    /*-----------------------------------*/
    /* for each slave in the system table */
    /*      send it a ServerShutdownMsg    */
    /*-----------------------------------*/
    for (int i = 0; i < slaverm.NumSlaves; i++)
        {
        if (i == 0) slaverm.FindFirst(sinfo);
        else slaverm.FindNext(sinfo);

        MakeRequest(sinfo.getSID(), request, reply);
        }

    return GOOD;
}
```

```
/*
    clientrm.C
    Client Resource Management.
    Brian Cox.
*/


#include "clientrm.h"


// static data members
unsigned long ClientResMgr::client_id = 0;

/*--------------------------------------------*/
/* add a client requester to the client table. */
/*--------------------------------------------*/
int ClientResMgr::Attach (ClientInfo &cinfo)
{
    int rc;

    cinfo.setCID(++client_id);    /* set client id */
    rc = Insert(cinfo);

    return rc;
}

/*-------------------------------------------------*/
/* remove a client requester from the client table. */
/*-------------------------------------------------*/
int ClientResMgr::Detach (ulong cid)
{
    ClientInfo cinfo;

    cinfo.setCID(cid);
    return SystemTable<ClientInfo>::Delete(cinfo);
}

/*----------------------------------------------*/
/* Find the entry for a particular Client id #. */
/*----------------------------------------------*/
int ClientResMgr::Find (ulong cid, ClientInfo &cinfo)
{
    ClientInfo c;
    int rc;

    c.setCID(cid);
    rc = SystemTable<ClientInfo>::Find(c);
    if (rc == GOOD)
        cinfo = c;

    return rc;
}
```

```
/*
     procrm.C
     Functions dealing with the Process System Table.
     Brian Cox.
*/

#include "procrm.h"

/* identifier for processes in the system. */
unsigned long ProcessResMgr::process_id = 0;

/*-------------*/
/* Constructor */
/*-------------*/
ProcessResMgr::ProcessResMgr()
{
     NumProcesses = 0;
}

/*-----------------------------------------------*/
/* allocate a slave on which to run a process.   */
/* look up the slave table for the next free slave */
/* and use that!                                 */
/*-----------------------------------------------*/
int ProcessResMgr::Alloc (ProcessInfo &pinfo)
{
     int rc;

     debug("ProcessResMgr::Alloc()");

     if (pinfo.getSID() == 0)
          {
          syserr("SID field is invalid, no process allocated!");
          return FAILED;
          }

     /*
     ;  pinfo.pid will be non zero if it is a process that is
     ;  being reallocated, so we must keep the old pid.
     */
     if (pinfo.getPID() == 0)
          {
          pinfo.setPID(++process_id);  /* this is a kludge - fix
later */

          /*
          ;  update the process table. Might be better if we wait
until
          ;  the slave has responded but causes trouble with
srvRunProcess().
          */
          rc = Insert(pinfo);
          if (rc == GOOD)
               NumProcesses++;  // add one to num of current
processes
          }

     return rc;
}


/*-----------------------------------------------*/
/* A Process is finished and the result has been */
/* accepted by a Client so delete it from the    */
/* Process System Table.                         */
/*                                               */
/* When should the process record be deleted?    */
/*  *  once the result has been entered in the   */
/*     result table, no                          */
/* or                                            */
/*  *  once the client has accepted the result.  */
/*-----------------------------------------------*/
int ProcessResMgr::Dealloc (ulong pid)
{
     ProcessInfo pinfo;
     int rc;

     debug("ProcessResMgr::Dealloc()");

     NumProcesses--;  // one less process to worry about

     /* now delete the process entry */
     pinfo.setPID(pid);
     rc = SystemTable<ProcessInfo>::Delete(pinfo);

     return rc;
}


/*-----------------------------------------------*/
/* Find the entry for a particular Process id #. */
/*-----------------------------------------------*/
int ProcessResMgr::Find (ulong pid, ProcessInfo &pinfo)
{
     ProcessInfo p;
     int rc;

     p.setPID(pid);
     rc = SystemTable<ProcessInfo>::Find(p);
     if (rc == GOOD)
          pinfo = p;

     return rc;
}
```

```
/*
     resultrm.C
     Result resource management.
     Brian Cox.
*/

#include "resultrm.h"
//#include "procrm.h"

/*--------------------------------------------------------*/
/*  The fiddling around might not be necessary if the result */
/*  is handled 'properly' by the Server.                  */
/*--------------------------------------------------------*/
int ResultResMgr::Add (ResultInfo rinfo)
{
     int rc;

     rc = Insert(rinfo);

     return rc;
}

/*--------------------------------------*/
/*  Delete a result from the results table. */
/*--------------------------------------*/
int ResultResMgr::Delete (ulong pid)
{
     int rc;
     ResultInfo rinfo;

     rinfo.setPID(pid);
     rc = SystemTable<ResultInfo>::Delete(rinfo);

     return rc;
}

/*---------------------------------------------*/
/* Get the entry for a particular Result based  */
/* on a process id #.                           */
/*                                              */
/* N.B.   ResultInfo is an unknown size as      */
/*        the result buffer is a pointer!       */
/*        Have to do some fiddling about.       */
/*---------------------------------------------*/
int ResultResMgr::Find (ulong pid, ResultInfo &rinfo)
{
     ResultInfo r;
     int rc;

     r.setPID(pid);
     rc = SystemTable<ResultInfo>::Find(r);
     if (rc == GOOD)
          rinfo = r;

     return rc;
}
```

```
/*
     slaverm.C
     Functions dealing with the Slave Resource Manager.
     Brian Cox.
*/

#include "slaverm.h"

/* Identifier for slaves in the system. */
ulong SlaveResMgr::slave_id = 0;


/*-------------*/
/* Constructor */
/*-------------*/
SlaveResMgr::SlaveResMgr()
{
     NumSlaves = 0;
}

/*------------------------------------------------*/
/*  A Slave Requester has attached to the Server */
/*  so add it to the Slave System Table.          */
/*------------------------------------------------*/
int SlaveResMgr::Attach (SlaveInfo &sinfo)
{
     int rc;

     /* set the slave id # */
     sinfo.setSID(++slave_id);          /* kludge - fix it later */
     sinfo.setStatus(SLAVE_RUNNING);

     rc = Insert(sinfo);
     if (rc == GOOD)
          NumSlaves++;

//   Loads[NOLOAD].AppendEntry(slave_id);

     return rc;
}


/*----------------------------------------------------*/
/*  A Slave Requester has disconnected from the       */
/*  Server so delete it from the Slave System Table. */
/*----------------------------------------------------*/
int SlaveResMgr::Detach (ulong sid)
{
     SlaveInfo sinfo;

     NumSlaves--;
//   Loads[NOLOAD].Delete(sid);
     sinfo.setSID(sid);
     return Delete(sinfo);
}


/*----------------------------------------------------*/
/* Select which Slave will run the next process.      */
/* Select the next slave in the table and update it. */
/* Round robin scheduler.                             */
/*----------------------------------------------------*/
int SlaveResMgr::Alloc (ulong &sid)
{
     SlaveInfo sinfo;
     int rc, i;

     // go to curr_slave's position in the table
     Find(curr_slave, sinfo);

     // now loop until we find a slave
     for (i=0; i < NumSlaves; i++)
          {
          //
          // if no more slaves left go back to the start of the
table.
          //
          if (FindNext(sinfo) == FAILED)
               if (FindFirst(sinfo) == FAILED)
                    {
                    // something wrong with system tables!
                    sid = INVALID_DPID;
                    syswarn("no slave available.");
                    return FAILED;
     }

          //
          // Find slave info and make sure it's not dead or
unavailable!
          //
          if (sinfo.getStatus() == SLAVE_RUNNING)
               {
               sid = sinfo.getSID();
               sinfo.setNumProcs(sinfo.getNumProcs()+1);
               return Update(sinfo);
               }
          }

     sid = INVALID_DPID;
     syswarn("no slave available.");
     return FAILED;
}


/*------------------------------*/
/* De-allocate a slave process */
/*------------------------------*/
int SlaveResMgr::Dealloc (ulong sid)
{
     SlaveInfo sinfo;
     int rc;

     if ((rc = Find(sid, sinfo)) == GOOD)
          {
          sinfo.setNumProcs(sinfo.getNumProcs()-1);
          rc = Update(sinfo);
          }

     return rc;
}


/*----------------------------------------*/
/* Find the entry for a particular Slave. */
/*----------------------------------------*/
int SlaveResMgr::Find (ulong sid, SlaveInfo &sinfo)
{
     SlaveInfo s;
     int rc;

     s.setSID(sid);
     rc = SystemTable<SlaveInfo>::Find(s);
     if (rc == GOOD)
          sinfo = s;

     return rc;
}
```

```
/*
     waitrm.C
     Brian Cox.
     System tables for both WaitProcess and WaitAnyProcess
requests.
*/


#include "waitrm.h"

/*------------------------------------------------*/
/* Add a WaitAnyProcess request to the table entry. */
/*------------------------------------------------*/
int WaitResMgr::Add (ulong pid, int sock)
{
     int rc;
     WaitInfo w;

     w.setPID(pid);
     w.setSock(sock);
     rc = Insert(w);

     return rc;
}

/*---------------------------------------------*/
/* Get the entry for a particular Wait request. */
/*---------------------------------------------*/
int WaitResMgr::Get (unsigned long pid, int &sock)
{
     int rc;
     WaitInfo w;

     w.setPID(pid);
     rc = Find(w);                 // find entry in the table
     if (rc == GOOD)
          {
          sock = w.getSock();
          rc = Delete(w);          // delete entry from table
          }

     return rc;
}
```

8.3 DistProc Slave

```
/*
	dpslave.C
	Brian Cox
	September 1993

	Distributed Processing Slave - main module.
*/

#define debug_on

#include <stdio.h>
#include <iostream.h>
#include <string.h>
#include <stdlib.h>
#include <signal.h>
#include "distproc.h"
#include "tcp.h"						/* network
communications */
#include "messages.h"					/* message objects
*/
#include "slvhndls.h"					/* slave message
handlers */


/*--------------*/
/* Constructor */
/*--------------*/
DPSlave *dpslv = NULL;


/*----------------------------------*/
/* termination and signal handler. */
/*----------------------------------*/
void DPSlaveShutDown(int sig)
{
	if (dpslv == NULL)
		cout << "\nDistProc Slave interrupted. Closing down.\n"
<< flush;
	else
		{
		cout << "\nDistProc Slave v" << dpslv->Version << "
closing down...\n"
			<< flush;
		dpslv->SlaveOK = FALSE;
//		delete dpslv;
		}

//	exit(1);
}


/*-----------------------*/
/* StartWatchDog Process */
/*-----------------------*/
//int StartWatchDog ()
//{
//	int pid;
//
//	pid = fork();
//	if (pid == -1)		// error in fork
//		{
//		cerr << "\nCouldn't fork watchdog process.\n" << flush;
//		dpslv->SlaveOK = FALSE;
//		return FAILED;
//		}
//	else if (pid != 0)	// parent process continously monitors
DPServer
//		{
//		for (i=
//		ClosePipe();	// RemProc socket
//		for (;;)
//			{
//			sleep(WATCHDOG_INTERVAL);
//			if (CheckServerStatus() == FAILED)
//				{
//				SendDummySrvShutDownMsgToSlave();
//				exit (1);
//				}
//			}
//		}
//	else				// child process
//		{
//
//		}
//}


/*-----------------------------------------------------------------
*/
/** Function to handle incoming requests from a DistProc Server.
*/
/*-----------------------------------------------------------------
*/
void HandleRequests()
{
	int newsock;

	debug("SlaveHandleRequests()");

	do
	{
		debug("Waiting for a request...");
		if (dpslv->WaitRequest(newsock) > 0)
			dpslv->DispatchRequest(newsock);
	}
	while (dpslv->SlaveOK == TRUE);
}


/*--------------*/
/* InitDPSlave */
/*--------------*/
int InitDPSlave (int argc, char *argv[])
{
	signal(SIGINT, DPSlaveShutDown);			/* signal handler
*/
	signal(SIGTERM, DPSlaveShutDown);

	if (argc != 2)
		{
		cerr << "\nError: no DPServer host name given.\n" <<
flush;
		exit(1);
		}

	dpslv = new DPSlave(argv[1]);
	if (dpslv == NULL)
		{
		cerr << "\nCouldn't create DPSlave.\n" << flush;
		exit(2);
		}
	if (dpslv->Registered() == FAILED)
		{
		cerr << "\nCouldn't register with DPServer.\n" << flush;
		delete dpslv;
		exit(3);
		}

	cout << "\nDistProc Slave v" << dpslv->Version << "
started.\n" << flush;
	return GOOD;
}


/*-----------------*/
/* main function. */
/*-----------------*/
int main(int argc, char *argv[])
{
	InitDPSlave(argc, argv);

//	StartWatchDog();

	HandleRequests();

	delete dpslv;

	return GOOD;
}
```

```
/*
;    slvhndls.C
;
;    By:       Brian Cox
;    Written:  Sept '93
;    Updated:  March '94
;    Use:      Distributed Processing Slave - message handler
functions.
*/

#define debug_on

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>
#include <iostream.h>
#include <sys/wait.h>
#include <sys/m_wait.h>
#include <sys/time.h>
#include <sys/select.h>
#include "distproc.h"                          /* system network
constants */
#include "tcp.h"                               /* network
communications   */
#include "messages.h"                          /* message objects
*/
#include "slvhndls.h"

/*----------------------------------*/
/* DistProc Slave global variables. */
/*----------------------------------*/
#define MAXRETRY  60
const char * const DPSlave::Version = "0.95";    /* slave version
number. */
unsigned long DPSlave::SlaveID = 0;
char * DPSlave::DPServerHostName;
int DPSlave::slavesock;

/*-------------*/
/* Constructor */
/*-------------*/
DPSlave::DPSlave(char * srvname)
{
     int i, rc;

     registered = FAILED;    // pessimistic approach
     SlaveOK = FALSE;
     SlaveID = 0;

     //
     // place the slave server in its own, private process group.
     // process group id is set to pid.
     //
     setpgrp();

     if (srvname != NULL)
          //
          // create the well known slave socket to which the
          // new process(RemProc) can attach before we spawn it
          // just to be safe. Avoids trouble with socket in use
          // timeouts also.
          //
          if (MakePipe(&SlaveRemProcSock, SLAVE_REMPROC_TCP_PORT,
5) == 0)
               //
               // try and open a connection with the dp server
               //
               for (i=0;i<MAXRETRY;i++)
                    if (OpenPipe(&slavesock, SERVER_TCP_PORT,
srvname) == GOOD )
                         [
                         if (ConnectToServer(srvname) == GOOD)
                              {
                              registered = GOOD;
                              SlaveOK = TRUE;
                              signal(SIGCLD,
DPSlave::DeathOfChildHandler);

                              // Stuff for select()
                              nfds = getdtablesize();
                              FD_ZERO(&afds);
                              FD_SET(slavesock, &afds);
                              }
                         else
                              syserr("Could not connect to DistProc
Server");

                         break;  // break out of retry!
                         }
                    else
                         {
                         syswarn("cannot open socket to DPServer -
still trying");
                         sleep(10);
                         }

          else
               syserr("SlaveRemProc socket already in use.");
     else
          syserr("No DistProc Server host name given.\n");
}

/*------------*/
/* Destructor */
/*------------*/
DPSlave::~DPSlave()
{
     if (Registered() == GOOD)
          {
          ClosePipe(SlaveRemProcSock);
          DisconnectFromServer();

          //
          // Close the original socket.
          //
          ClosePipe(slavesock);

          if (DPServerHostName != NULL)
               delete [] DPServerHostName;

          cout << endl << flush;
          }
}

/*--------------------*/
/* Wait for a request */
/*--------------------*/
int DPSlave::WaitRequest (int &sock)
{
     int clilen, rc, fd;

     if (Registered() == FAILED)
          return -1;

     while (1)
          {
          tout.tv_sec = 40;  // times out after 40 seconds.
          tout.tv_usec = 0;
          bcopy((char *)&afds, (char *)&rfds, sizeof(rfds));
          rc = select(nfds, &rfds, NULL, NULL, &tout);

          //
          // Make sure select returned OK
          //
          if (rc < 0)
               {
               syserr("select error");
               return rc;
               }

          else
               //
               // Check if select timed out
               //
               if (rc == 0)
                    {
                    cout << "\nSelect timed out!!!\n" << flush;
                    return rc;
                    }

          else
               //
               // We have activity on a socket.
               //
               if (rc > 0)
                    {
                    sock = slavesock;
                    return rc;
                    }
          } // end while

     return rc;
}

/*-------------------------------------------------------------------
*/
/* For the moment this is a serialised Slave. It probably
*/
/* should fork a process to handle the incoming request.
*/
/* The reasons for forking() a sub process to handle the
*/
/* servers request are:
*/
/*
*/
/*   1.   multiple DistProc Servers - several requests
*/
/*
*/
/*   2.   DistProc message from server which effects either
*/
/*        the Slave process (ClientShutDown or ServerShutDown) or
*/
/*        a sub process it is executing on behalf of the DistProc
*/
/*        Server (e.g. KillProcess)
*/
/*-------------------------------------------------------------------
*/
int DPSlave::DispatchRequest (int sock)
{
     Service sv = no_service;
     Buffer b;                          /* pointer to message buffer
*/
     unsigned len;                      /* length of message */
     BaseMsg msg;                       /* base message object */
     int rc;

     debug("DispatchRequest()");

     if (Registered() == FAILED)
          return FAILED;

     //
     // read the request from the pipe
     //
     if (ReadPipe(sock, (void **)&b.data, (unsigned long
*)&b.length, 60) == FAILED)
          {
          syserr("couldn't read request from DistProc Server");
          SlaveOK = FALSE;   // assume that the Server is dead and
exit
          return FAILED;
          }

     msg.load(b);                       /* recreate the message object
*/
     sv = msg.getService();             /* what's the request? */

     switch (sv)
          {
          //
          // The following messages can only come from a DistProc
Server.
          //
          case run_proc        : rc = RunProcess(sock, b);
```

```
                                  break;

          case kill_proc        : rc = KillProcess(sock, b);
                                  break;

          case server_shutdown  : rc = ServerShutDown(sock, b);
                                  break;

//        case check_slave      : rc = CheckSlave(sock, b);
//                                break;

          default : syswarn("DistProc Slave: unknown message
received.");
                    break;
          }

//   ClosePipe(sock);

     return GOOD;
}

/*============================================================*/
/* The following functions handle incoming requests from a */
/* DistProc Server.  They take care of all Server -> Slave */
/* requests.                                                */
/*============================================================*/

/*--------------------------------------------------------------*/
/* Handle a RUN_PROCESS request from a server.                  */
/* Have to pass along the output address and outsize also!     */
/* The actual size of the pointer to output must be passed     */
/* around as well as pointers will vary in size across         */
/* various platforms and systems.  It's ok for the moment      */
/* as UNIX is the only OS in use.                               */
/*--------------------------------------------------------------*/
int DPSlave::RunProcess(int s, const Buffer &b)
{
     RunProcessMsg msg(b);
     Buffer request, response;
     char clientname[MAX_LEN_CLIENT_NAME],
filename[MAX_LEN_IMAGENAME];
     void * input;
     unsigned insize;
     int actual_pid = 0;
     int rc;

     if (Registered() == FAILED)
          return FAILED;

     debug("RunProcess()");

     //
     // Extract information from the RunProcessMsg message
     //
     msg.pinfo.getClientName(clientname);
     msg.pinfo.getFileName(filename);
     msg.pinfo.getInput(input, insize); /* get the input for the
process */

     //
     // fork the given process under the original users account.
     //
     rc = SpawnProcess(clientname, filename,
                       input, insize,
                       msg.pinfo.getPID(), actual_pid);

     //
     // fill out the return fields in the message and
     // return it to the server.
     //
     msg.retcode = rc;                    /* rc of SpawnProcess */
     msg.pinfo.setActualPID(actual_pid); /* return actual pid of
new process */

     msg.save(request);

     if (FulfillRequest(s, request) == FAILED)
          {
          syswarn("sending RunProcess reply message to server");
          return(FAILED);
          }

     delete [] input;

     return rc;
}

/*------------------------------------------------*/
/* handle a KILL_PROCESS request from the Server. */
/*------------------------------------------------*/
int DPSlave::KillProcess(int s, const Buffer &b)
{
     KillProcessMsg msg(b), rmsg;
     Buffer response;
     int pid, status;

     if (Registered() == FAILED)
          return FAILED;

     debug("Kill Process");

     //
     // get pid of process to kill.
     //
     pid = msg.pinfo.getActualPID();

     //
     // delete the pid from the list of children and then
     // kill of the child process.
     //
     status = KillChild(pid);

     //
     // send back the result of wait to the server.
     //
     msg.retcode = status;
     msg.save(response);
     if ( FulfillRequest(s, response) == FAILED)
          {
          syswarn("couldn't send Kill reply message to server");
          return FAILED;
          }

     //
     // see if the process actually existed.
     //
     if ( status == -1 )
          {
          syswarn("process does not exist");
          return(FAILED);
          }

     return(GOOD);
}

/*-------------------------------------------------------*/
/* The Server is closing down.  The Slave must kill off */
/* it's child processes and exit.  No reply message.    */
/*-------------------------------------------------------*/
int DPSlave::ServerShutDown(int s, const Buffer &b)
{
     BaseMsg msg(b);
     Buffer reply;

     if (Registered() == FAILED)
          return FAILED;

     debug("DPSlave::ServerShtuDown");

     msg.retcode = GOOD;
     msg.save(reply);
     if (FulfillRequest(s, reply) == FAILED)
          {
          syswarn("sending serverShutDown reply message to
server");
          return(FAILED);
          }

     //
     // kill off all the children.
     //
     KillAllChildren();
     SlaveOK = FALSE;   // set flag to show that slave must
shutdown!

     return GOOD;
}


/*--------------------------------------*/
/* check if connection was successful. */
/*--------------------------------------*/
int DPSlave::Registered()
{
     return registered;
}


/*===========================*/
/* Protected member functions. */
/*===========================*/

/*--------------------------------------------------------------*/
/* open a connection with the server, send the request message */
/* and then read the response from the server.  Finally close  */
/* the connection with the server.                              */
/*--------------------------------------------------------------*/
int DPSlave::MakeRequest(const Buffer &request, Buffer &response)
{
     int i;

     debug("DPSlave::MakeRequest()");

     //
     // perform a request/response transaction, i.e. send the
request
     // to the server and then wait to read take the response back
     // from the server.
     //
     if ( TransactPipe(slavesock, request.data, request.length,
                       (void **)&response.data,
                       (unsigned long *)&response.length, 180) !=
GOOD )
          syswarn("in pipe transaction");

     return GOOD;
}

/*-----------------------------------------------------*/
/* send back the filled in message to the server.      */
/* The reason for the name change in the 2nd parameter */
/* is that to fulfill a request from a server          */
/* a slave must return a response - this is it.        */
/*-----------------------------------------------------*/
int DPSlave::FulfillRequest(int sock, const Buffer &response)
{
     debug("DPSlave::FulfillRequest()");

     //
     // Write the message to the server.
     //
     if ( WritePipe(sock, response.data, response.length) != GOOD
)
          {
          syswarn("writing to pipe");
          return(FAILED);
          }

     return(GOOD);
}


/*----------------------------------------------------------*/
/* connect to the server to announce our engagement.        */
/*                                                          */
/* The following aren't used at the moment!                 */
/*                                                          */
/*  1.  what must the slave send to the server?             */
/*          requester type - slave                          */
/*          slave address. the server can get this itself! */
/*                                                          */
/*  2.  what must the server send back to the slave?        */
/*          slave id                                        */
/*----------------------------------------------------------*/
int DPSlave::ConnectToServer(char * srvname)
{
     SlaveConnectMsg msg, rmsg;
     Buffer request, response;

     debug("DPSlave::ConnectToServer()");

     if (srvname == NULL)
          {
          syswarn("no hostname specified for DistProc Server");
          return FAILED;
          }

     DPServerHostName = new char[strlen(srvname)+1];
     strcpy(DPServerHostName, srvname);

     //
     // set the source address field of the message.
     //
     gethostname(SlaveHostName, MAXHOSTNAMELEN);
     msg.sinfo.setHostName(SlaveHostName);
     msg.sinfo.setPort(SLAVE_TCP_PORT);

     //
     // Inform the server of our presence and that we are a slave.
     // make the request to the server.
     //
     msg.save(request);
     if (MakeRequest(request, response) == FAILED)
          {
          syswarn("couldn't send Connect message to server");
          return FAILED;
          }

     //
     // reconstruct the response message into a message object.
     //
     rmsg.load(response);
     if ( rmsg.retcode == FAILED )
          {
          syswarn("DistProc Server wouldn't accept slaves
connection request");
          return(FAILED);
          }

     //
     // extract Slave-id from the response message.
     //
     SlaveID = rmsg.sinfo.getSID();

     return GOOD;
}

/*----------------------------------------------------------*/
/* Disconnect from the server.  This will kill off any      */
/* child processes that are being executing on behalf       */
/* of the server.                                           */
/*----------------------------------------------------------*/
int DPSlave::DisconnectFromServer()
{
     SlaveDisconnectMsg msg, rmsg;
     Buffer request, response;

     if (Registered() == FAILED)
          return FAILED;

     debug("DPSlave::DisconnectFromServer()");

     //
     // set flag to show that Slave is closing down!
     //
     SlaveOK = FALSE;

     //
     // let server know who we are.
     //
     msg.setSystemID(SlaveID);

     //
     // tell the server that we're closing down.
     // make the request to the server.
     //
     msg.save(request);
     if (MakeRequest(request, response) == FAILED)
          {
          syswarn("sending Disconnect message to server");
          return FAILED;
          }

     //
     // kill off all the slaves children.
     //
     KillAllChildren();

     rmsg.load(response);

     return rmsg.retcode;
}

/*****************************/
/* Private member functions */
/*****************************/

/*------------------------------------------------------------------*/
/* Create a process on behalf of the Server.                        */
/* As well as forking() the appropriate process other               */
/* information so that the new process can:                         */
/*   1.  access it's input                                          */
/*   2.  communicate with it's parent (Slave) so that the           */
/*       output of the process can be returned to the client.       */
/* How is this info passed?                                         */
/*------------------------------------------------------------------*/
int DPSlave::SpawnProcess(char *clientname, char *filename,
                          void *input, unsigned int insize,
                          unsigned long pid, int &actual_pid)
{
     int s, clilen;
     struct sockaddr_in cli_addr;
     int rc, childpid;
     char account[MAX_LEN_CLIENT_NAME+3] = "-l ";  // field for
clients name

     if (Registered() == FAILED)
          return FAILED;

     debug("DPSlave::SpawnProcess()");

     strcat(account, clientname);

     //
     // try and create the new process.
     //
     if ( (childpid = fork()) == -1 )
          {
          syserr("cannot fork");
          return(FAILED);
          }
     else if (childpid == 0)      // ------------  Child Process  -
----------
          {
          debug("\tChild process - calling execlp() ...");
          cout << "\nIn forked process; slave socket #" <<
slavesock << endl;
          ClosePipe(slavesock);
          ClosePipe(SlaveRemProcSock);

          //
          // This one uses security...
          //
//        rc = execlp("rsh", "rsh", SlaveHostName, "-l",
clientname, "nice", filename, NULL);

          //
          // This one doesn't use security...
          //
          rc = execlp(filename, filename, NULL);
          syserr("\tcouldn't execlp() process!");

          /* ClosePipe(paramsock); DO I NEED TO CLOSE THE PIPE IN
THE CHILD ? */
          //
          // Need to inform Slave that process wasn't executed
correctly
          // before we exit!!
          //
          exit(rc);
          }
     else      // ----------  Parent Process  -----------
          {
          debug("\tRemProc process spawned!  This is the Slave
process.");
          cout << "\n\tRemProc system pid is " << childpid << endl
<< flush;

          //
          // SHOULD DO A TIMEOUT HERE IN CASE THE CHILD PROCESS
          // NEVER CONTACTS US, CAUSING A DEADLOCK IN THE DPSLAVE
          //

          //
          // wait for the new process to contact us to look for
          // its input and output parameters.
          //
          rc = WaitConnectPipe(SlaveRemProcSock, &s, &cli_addr,
&clilen);

          if (rc != GOOD)
               syserr("waiting for RemProc to connect to the
Slave!");
          else
               {
               //
               // send the processes DistProc system PID.
               // send this Slaves id #.
               // send the input parameters.
               //
               WritePipe(s, (void *)&pid, sizeof(pid));
               WritePipe(s, (void *)&SlaveID, sizeof(SlaveID));
               WritePipe(s, (void *)DPServerHostName,
strlen(DPServerHostName)+1);
               WritePipe(s, input, insize);
               }

          ClosePipe(s);

          actual_pid = childpid;
          return rc;
          }

     //
     // something seriously wrong if we reach here!
     //
     syserr("something seriously wrong with fork()!!");

     return FAILED;
}

/*--------------------------------------------------------------*/
/* send a process termination signal to the specified           */
/* child process and then wait for the SIGCLD (Death of child) */
/* signal to show that the process is terminated.               */
/*--------------------------------------------------------------*/
int DPSlave::KillChild(int pid)
{
     int status;

     debug("DPSlave::KillChild()");

     //
     // we don't want to the DOC handler to report to the DPServer
     //
     signal(SIGCLD, SIG_IGN);

     status = kill(pid, SIGKILL);     /* child can't catch SIGKILL
signal */
```

```
	if ( status == 0 )
		pid = wait(&status);		// -1 => no more child
processes
	else
		{
		syserr("DPSlave::KillChild: couldn't kill process");
		cerr << "Error: couldn't kill process #" << pid << endl
<< flush;
		}

	//
	// reestablish the DOC signal handler for future children
	//
	signal(SIGCLD, DPSlave::DeathOfChildHandler);

	return status;
}

/*-----------------------------------*/
/* kill off all the slaves children. */
/*-----------------------------------*/
void DPSlave::KillAllChildren()
{
	int status, grppid;

	debug("DPSlave::KillAllChildren()");

	grppid = getpgrp();		/* get this process groups pid
*/

	//
	// we don't want to the DOC handler to report to the DPServer
	//
	signal(SIGCLD, SIG_IGN);

	//
	// send a SIGTERM signal to each child
	//
	status = kill(-(grppid), SIGKILL);
	if ( status == 0 )
		while (wait(&status) > 0)  // -1 => no more child
processes

	//
	// reestablish the DOC signal handler for future children
	//
	signal(SIGCLD, DPSlave::DeathOfChildHandler);
}

/*---------------------------------------------------------------*/
/* Catches DOC signals.  If the child core dumped then we must */
/* inform the DistProc Server of the childs untimely demise.   */
/*---------------------------------------------------------------*/
void DPSlave::DeathOfChildHandler (int sig)
{
	int pid, childpid;
	union wait status;
	char temp[9];

	debug("DPSlave::DeathOfChildHandler()");

	//
	// reestablish the DOC signal handler for future children
	//

	pid = wait((int *)&status);

	cout << "\nRemProc #" << pid << ", exit status " <<
status.w_status << "\n" << flush;

	cout << "\nwait() returned with pid #" << pid
	     << "\nw_status: " << status.w_status
	    << " \nw_retcode: " << status.w_retcode
	    << " \nw_coredump: " << status.w_coredump
	    << " \nw_termsig: " << status.w_termsig;

	// Only inform the DPServer if a Child core dumped!
	if (status.w_coredump != 0)
		if ( (childpid = fork()) == -1 )
			syserr("cannot fork in DeathOfChildHandler");
		else if (childpid == 0)
			//
			// child informs Server that a RemProc core dumped.
			//
			{
			DOCMsg msg;
			Buffer request, reply;

			msg.sid = SlaveID;
			msg.realpid = pid;
			msg.save(request);
			debug("DPSlave::DOCHandler: making request to
DPServer");
			if (MakeRequest(request, reply) == FAILED)
				syserr("sending DOC message to server");
			debug("DPSlave::DOCHandler: request completed,
exiting");

			exit(0);
			}

	//
	// Parent continues.
	//

	signal(SIGCLD, DPSlave::DeathOfChildHandler);
}
```

8.4 Message Classes

```
/*
     messages.C
     Brian Cox.
     August 1993.

     All the system message class manipulator functions are in
this file.
*/

#include "msgdefs.h"
#include "messages.h"
#include "saveobj.h"

/*==============================================================*/
/*                       B a s e M s g                          */
/*==============================================================*/

// BaseMsg constructor
BaseMsg::BaseMsg()
{
     length = 0;
     retcode = GOOD;
     setService(no_service);
     setSystemID(0);
}

// BaseMsg re-constructor
BaseMsg::BaseMsg(Buffer b)
{
     load(b);
}

// doesn't work if pointers are used - only arrays!
int BaseMsg::save(Buffer &b)
{
     // first of all make sure b is empty
     b.empty();

     // now copy each element of the class to the buffer
     SAVE(b, length);
     SAVE(b, retcode);
     SAVE(b, system_id);
     SAVE(b, sv);

     return b.length;
}

Buffer BaseMsg::load(const Buffer& b)
{
     Buffer tmp;

     if (b.length <= 0 || b.data == NULL)
          {
          syserr("BaseMsg::load() passed an empty message
buffer");
          tmp.empty();
          return tmp;
          }

     tmp = LOAD(b, length);
     tmp = LOAD(tmp, retcode);
     tmp = LOAD(tmp, system_id);
     tmp = LOAD(tmp, sv);

     return tmp;    // points to rest of object in buffer - derived
classes.
};

/*==============================================================*/
/*                         D O C M s g                          */
/*==============================================================*/

/*-------------*/
/* constructor */
/*-------------*/
DOCMsg::DOCMsg()
          : BaseMsg()
{
     setService(death_of_child);
}


/*---------------*/
/* re-constructor */
/*---------------*/
DOCMsg::DOCMsg(Buffer b)
                    : BaseMsg()
{
     load(b);
}


/*---------------------*/
/* DOCMsg::save */
/*---------------------*/
int DOCMsg::save(Buffer &b)
{
     BaseMsg::save(b);

     SAVE(b, sid);
     SAVE(b, realpid);

     return b.length;
}


/*---------------------*/
/* DOCMsg::Load */
/*---------------------*/
Buffer DOCMsg::load(const Buffer& b)
{
     Buffer tmp;

     tmp = BaseMsg::load(b); // b points to rest of object in
buffer
     tmp = LOAD(tmp, sid);
     tmp = LOAD(tmp, realpid);

     return tmp;
}


/*==============================================================*/
/*           S e r v e r S h u t D o w n M s g                  */
/*==============================================================*/

/*-------------*/
/* constructor */
/*-------------*/
ServerShutDownMsg::ServerShutDownMsg()
                    : BaseMsg()
{
     setService(server_shutdown);
}

/*----------------*/
/* re-constructor. */
/*----------------*/
ServerShutDownMsg::ServerShutDownMsg(Buffer b)
                    : BaseMsg()
{
     load(b); // this will fill in BaseMsg by calling
BaseMsg::load()
}

/*-----------------------*/
/* ServerShutDownMsg::save */
/*-----------------------*/
int ServerShutDownMsg::save(Buffer &b)
{
     BaseMsg::save(b);        // call parent objects save() member
function

     return b.length;
}

/*-----------------------*/
/* ServerShutDownMsg::load */
/*-----------------------*/
Buffer ServerShutDownMsg::load(const Buffer& b)
{
     return BaseMsg::load(b);        // b points to rest of object
in buffer
}


/*==============================================================*/
/*             C l i e n t C o n n e c t M s g                  */
/*==============================================================*/

/*-------------*/
/* constructor */
/*-------------*/
ClientConnectMsg::ClientConnectMsg()
                    : BaseMsg()
{
     setService(client_connect);
}

/*----------------*/
/* re-constructor. */
/*----------------*/
ClientConnectMsg::ClientConnectMsg(Buffer b)
                    : BaseMsg()
{
     load(b); // this will fill in BaseMsg by calling
BaseMsg::load()
}

/*-----------------------*/
/* ClientConnectMsg::save */
/*-----------------------*/
int ClientConnectMsg::save(Buffer &b)
{
     BaseMsg::save(b);        // call parent objects save() member
function
     cinfo.save(b);

     return b.length;
}

/*-----------------------*/
/* ClientConnectMsg::load */
/*-----------------------*/
Buffer ClientConnectMsg::load(const Buffer& b)
{
     Buffer tmp;

     tmp = BaseMsg::load(b); // b points to rest of object in
buffer
     tmp = cinfo.load(tmp);

     return tmp;
}

/*==============================================================*/
/*          C l i e n t D i s c o n n e c t M s g               */
/*==============================================================*/

/*-------------*/
/* constructor */
/*-------------*/
ClientDisconnectMsg::ClientDisconnectMsg()
                         : BaseMsg()
```

```
{
     setService(client_shutdown);
}

/*-----------------*/
/* re-constructor. */
/*-----------------*/
ClientDisconnectMsg::ClientDisconnectMsg(Buffer b)
                         : BaseMsg()
{
     load(b);
}


/*----------------------------*/
/* ClientDisconnectMsg::save */
/*----------------------------*/
int ClientDisconnectMsg::save(Buffer &b)
{
     BaseMsg::save(b);

     return b.length;
}


/*----------------------------*/
/* ClientDisconnectMsg::Load */
/*----------------------------*/
Buffer ClientDisconnectMsg::load(const Buffer& b)
{
     return BaseMsg::load(b);        // b points to rest of object
in buffer
}


/*=================================================================*/
/*                R u n P r o c e s s M s g                        */
/*=================================================================*/

/*-------------*/
/* constructor */
/*-------------*/
RunProcessMsg::RunProcessMsg()
{
     setService(run_proc);
}

/*----------------*/
/* re-constructor */
/*----------------*/
RunProcessMsg::RunProcessMsg(Buffer b)
                    : BaseMsg()
{
     load(b);
}

/*----------------------*/
/* RunProcessMsg::save */
/*----------------------*/
int RunProcessMsg::save(Buffer &b)
{
     BaseMsg::save(b);

     pinfo.save(b);

     return b.length;
}

/*----------------------*/
/* RunProcessMsg::load */
/*----------------------*/
Buffer RunProcessMsg::load(const Buffer& b)
{
     Buffer tmp;

     tmp = BaseMsg::load(b); // b points to rest of object in
buffer
     tmp = pinfo.load(tmp);

     return tmp;
}


/*=================================================================*/
/*                W a i t P r o c e s s M s g                      */
/*=================================================================*/

/*-------------*/
/* constructor */
/*-------------*/
WaitProcessMsg::WaitProcessMsg()
                         : BaseMsg()
{
     setService(wait_proc);
}


/*----------------*/
/* re-constructor */
/*----------------*/
WaitProcessMsg::WaitProcessMsg(Buffer b)
                         : BaseMsg()
{
     load(b);
}


/*-----------------------*/
/* WaitProcessMsg::save */
/*-----------------------*/
int WaitProcessMsg::save(Buffer &b)
{
     BaseMsg::save(b);

     pinfo.save(b);
     rinfo.save(b);

     return b.length;
}
```

```
/*-----------------------*/
/* WaitProcessMsg::Load */
/*-----------------------*/
Buffer WaitProcessMsg::load(const Buffer& b)
{
     Buffer tmp;

     tmp = BaseMsg::load(b); // b points to rest of object in
buffer
     tmp = pinfo.load(tmp);
     tmp = rinfo.load(tmp);

     return tmp;
}


/*=================================================================*/
/*                K i l l P r o c e s s M s g                      */
/*=================================================================*/

/*-------------*/
/* constructor */
/*-------------*/
KillProcessMsg::KillProcessMsg()
                    : BaseMsg()
{
     setService(kill_proc);
}


/*----------------*/
/* re-constructor */
/*----------------*/
KillProcessMsg::KillProcessMsg(Buffer b)
                    : BaseMsg()
{
     load(b);
}


/*-----------------------*/
/* KillProcessMsg::save */
/*-----------------------*/
int KillProcessMsg::save(Buffer &b)
{
     BaseMsg::save(b);

     pinfo.save(b);

     return b.length;
}


/*-----------------------*/
/* KillProcessMsg::Load */
/*-----------------------*/
Buffer KillProcessMsg::load(const Buffer& b)
{
     Buffer tmp;

     tmp = BaseMsg::load(b); // b points to rest of object in
buffer
     tmp = pinfo.load(tmp);

     return tmp;
}


/*=================================================================*/
/*                     R e s u l t M s g                           */
/*=================================================================*/

/*-------------*/
/* constructor */
/*-------------*/
ResultMsg::ResultMsg()
          : BaseMsg()
{
     setService(result);
}


/*----------------*/
/* re-constructor */
/*----------------*/
ResultMsg::ResultMsg(Buffer b)
          : BaseMsg()
{
     load(b);
}


/*-----------------------*/
/* ResultMsg::save */
/*-----------------------*/
int ResultMsg::save(Buffer &b)
{
     BaseMsg::save(b);

     rinfo.save(b);

     return b.length;
}


/*-----------------*/
/* ResultMsg::Load */
/*-----------------*/
Buffer ResultMsg::load(const Buffer& b)
{
     Buffer tmp;

     tmp = BaseMsg::load(b);
     tmp = rinfo.load(tmp);

     return tmp;
}
```

```
/*================================================================*/
/*                S l a v e C o n n e c t M s g                   */
/*================================================================*/

/*-------------*/
/* constructor */
/*-------------*/
SlaveConnectMsg::SlaveConnectMsg()
                : BaseMsg()
{
     setService(slave_connect);
}


/*----------------*/
/* re-constructor */
/*----------------*/
SlaveConnectMsg::SlaveConnectMsg(Buffer b)
                : BaseMsg()
{
     load(b);
}


/*---------------------*/
/* SlaveConnectMsg::save */
/*---------------------*/
int SlaveConnectMsg::save(Buffer &b)
{
     BaseMsg::save(b);

     sinfo.save(b);

     return b.length;
}


/*---------------------*/
/* SlaveConnectMsg::Load */
/*---------------------*/
Buffer SlaveConnectMsg::load(const Buffer& b)
{
     Buffer tmp;

     tmp = BaseMsg::load(b); // b points to rest of object in
buffer
     tmp = sinfo.load(tmp);

     return tmp;
}


/*================================================================*/
/*            S l a v e D i s C o n n e c t M s g                 */
/*================================================================*/

/*-------------*/
/* constructor */
/*-------------*/
SlaveDisconnectMsg::SlaveDisconnectMsg()
                    : BaseMsg()
{
     setService(slave_shutdown);
}


/*----------------*/
/* re-constructor */
/*----------------*/
SlaveDisconnectMsg::SlaveDisconnectMsg(Buffer b)
                    : BaseMsg()
{
     load(b);
}


/*-------------------------*/
/* SlaveDisconnectMsg::save */
/*-------------------------*/
int SlaveDisconnectMsg::save(Buffer &b)
{
     BaseMsg::save(b);

     return b.length;
}


/*-------------------------*/
/* SlaveDisconnectMsg::Load */
/*-------------------------*/
Buffer SlaveDisconnectMsg::load(const Buffer& b)
{
     return BaseMsg::load(b);        // b points to rest of object
in buffer
}


/*================================================================*/
/*                Q u e r y S t a t u s M s g                     */
/*================================================================*/

/*-------------*/
/* constructor */
/*-------------*/
QueryStatusMsg::QueryStatusMsg()
                : BaseMsg()
{
     setService(query_status);
}


/*----------------*/
/* re-constructor */
/*----------------*/
QueryStatusMsg::QueryStatusMsg(Buffer b)
                : BaseMsg()
{
     load(b);
}


/*---------------------*/
/* QueryStatusMsg::save */
/*---------------------*/
int QueryStatusMsg::save(Buffer &b)
{
     BaseMsg::save(b);

     return b.length;
}


/*---------------------*/
/* QueryStatusMsg::Load */
/*---------------------*/
Buffer QueryStatusMsg::load(const Buffer& b)
{
     return BaseMsg::load(b);        // b points to rest of object
in buffer
}
```

8.5 Info Classes

```
/*
     prcinfo.C
     Remote Process Information class member functions.
     Brian Cox.
*/

#include "prcinfo.h"

/*-------------*/
/* Constructor */
/*-------------*/
ProcessInfo::ProcessInfo ()
{
     setPID(0);
     setCID(0);
     setSID(0);
     setActualPID(0);
     setStatus(0);
     setProcRetCode(0);
     setClientName("");
     setFileName("");
     setOutputPtr((Buffer *)NULL);
//   result_ptr = NULL;

     input_buff = NULL;       // don't use setInput(NULL,0)
     insize = 0;
}

/*------------------*/
/* Copy Constructor */
/*------------------*/
ProcessInfo::ProcessInfo (const ProcessInfo &source)
{
     *this = source;
}

/*------------*/
/* Destructor */
/*------------*/
ProcessInfo::~ProcessInfo ()
{
     if (input_buff != NULL && insize > 0)
          delete [] input_buff;
}

/*---------------------*/
/* Assignment Operator */
/*---------------------*/
ProcessInfo& ProcessInfo::operator = (const ProcessInfo &source)
{
     // check if we are copying ourselves?
     if (this == &source)
          return *this;

     if (input_buff != NULL && insize > 0)
          delete [] input_buff;

     setPID(source.pid);
     setCID(source.cid);
     setSID(source.sid);
     setActualPID(source.actual_pid);
     setStatus(source.status);
     setProcRetCode(source.procretcode);
     setClientName((char *)source.clientname);
     setFileName((char*)source.filename);
     setInput(source.input_buff, source.insize);

     size_of_output_ptr = source.size_of_output_ptr;
     memcpy(output_ptr, source.output_ptr, MAX_PTR_SIZE);
//   result_ptr = source.result_ptr;

     return *this;
}

/*-------------------*/
/* Equality Operator */
/*-------------------*/
int ProcessInfo::operator == (const ProcessInfo &source)
{
     return (pid == source.pid);
}

/*-------------*/
/* Save Object */
/*-------------*/
int ProcessInfo::save (Buffer &b)
{
     // now copy each element of the class to the buffer
     SAVE(b, pid);
     SAVE(b, cid);
     SAVE(b, sid);
     SAVE(b, actual_pid);
     SAVE(b, status);
     SAVE(b, procretcode);
     SAVE(b, clientname, (size_t)MAX_LEN_CLIENT_NAME);
     SAVE(b, filename, (size_t)MAX_LEN_IMAGENAME);

     SAVE(b, size_of_output_ptr);
     SAVE(b, output_ptr, (size_t)MAX_PTR_SIZE);
//   SAVE(b, (&result_ptr), sizeof(Buffer *));

     SAVE(b, insize);
     if (insize > 0)
          SAVE(b, (char *)input_buff, (size_t)insize);

     return b.length;
}

/*-------------*/
/* Load Object */
/*-------------*/
Buffer ProcessInfo::load (const Buffer& b)
{
     Buffer tmp = b;

     tmp = LOAD(tmp, pid);
     tmp = LOAD(tmp, cid);
     tmp = LOAD(tmp, sid);
     tmp = LOAD(tmp, actual_pid);
     tmp = LOAD(tmp, status);
     tmp = LOAD(tmp, procretcode);
     tmp = LOAD(tmp, (char *)clientname,
(size_t)MAX_LEN_CLIENT_NAME);
     tmp = LOAD(tmp, (char *)filename, (size_t)MAX_LEN_IMAGENAME);

     tmp = LOAD(tmp, size_of_output_ptr);
     tmp = LOAD(tmp, output_ptr, (size_t)MAX_PTR_SIZE);
//   tmp = LOAD(tmp, (&result_ptr), sizeof(Buffer *));

     tmp = LOAD(tmp, insize);
     if (insize > 0)
          {
          input_buff = (void *)new char[insize];
          tmp = LOAD(tmp, (char *)input_buff, (size_t)insize);
          }
     else
          input_buff = NULL;

     return tmp;
}

/*---------------------*/
/* ProcessInfo::getPID */
/*---------------------*/
unsigned long ProcessInfo::getPID ()
{
     return pid;
}

/*---------------------*/
/* ProcessInfo::getCID */
/*---------------------*/
unsigned long ProcessInfo::getCID ()
{
     return cid;
}

/*---------------------*/
/* ProcessInfo::getSID */
/*---------------------*/
unsigned long ProcessInfo::getSID ()
{
     return sid;
}

/*----------------------------*/
/* ProcessInfo::getClientName */
/*----------------------------*/
void ProcessInfo::getClientName (char
clnname[MAX_LEN_CLIENT_NAME])
{
     strncpy(clnname, clientname, MAX_LEN_CLIENT_NAME-1);
     clnname[MAX_LEN_CLIENT_NAME-1] = '\0';
}

/*-------------------------*/
/* ProcessInfo::getFileame */
/*-------------------------*/
void ProcessInfo::getFileName (char fname[MAX_LEN_IMAGENAME])
{
     strncpy(fname, filename, MAX_LEN_IMAGENAME-1);
     fname[MAX_LEN_IMAGENAME-1] = '\0';
}

/*---------------------------*/
/* ProcessInfo::getActualPID */
/*---------------------------*/
int ProcessInfo::getActualPID ()
{
     return actual_pid;
}

/*------------------------*/
/* ProcessInfo::getStatus */
/*------------------------*/
int ProcessInfo::getStatus ()
{
     return status;
}

/*-----------------------------*/
/* ProcessInfo::getProcRetCode */
/*-----------------------------*/
int ProcessInfo::getProcRetCode ()
{
     return procretcode;
}

/*-----------------------*/
/* ProcessInfo::getInput */
/*-----------------------*/
void ProcessInfo::getInput (void * &input, unsigned &size)
{
     size = insize;
     input = (char *) new char [insize];
     memcpy(input, input_buff, insize);
}

/*-----------------------------------------------------*/
/* ProcessInfo::getOutputPtr                           */
/*                                                     */
/* Here, we want to get the saved pointers address and */
/* copy it to the address pointed to by outptr.        */
/*-----------------------------------------------------*/
void ProcessInfo::getOutputPtr (Buffer * outptr)
{
    memcpy(outptr, output_ptr, sizeof(outptr));
}

/*---------------------*/
/* ProcessInfo::setPID */
/*---------------------*/
void ProcessInfo::setPID (unsigned long procid)
{
    pid = procid;
}

/*---------------------*/
/* ProcessInfo::setCID */
/*---------------------*/
void ProcessInfo::setCID (unsigned long clientid)
{
    cid = clientid;
}

/*---------------------*/
/* ProcessInfo::setSID */
/*---------------------*/
void ProcessInfo::setSID (unsigned long slaveid)
{
    sid = slaveid;
}

/*----------------------------*/
/* ProcessInfo::setClientName */
/*----------------------------*/
void ProcessInfo::setClientName (char * clnname)
{
    strncpy(clientname, clnname, MAX_LEN_CLIENT_NAME);
    clientname[MAX_LEN_CLIENT_NAME-1] = '\0';
}

/*---------------------------*/
/* ProcessInfo::setFileName */
/*---------------------------*/
void ProcessInfo::setFileName (char * fname)
{
    strncpy(filename, fname, MAX_LEN_IMAGENAME);
    filename[MAX_LEN_IMAGENAME-1] = '\0';
}

/*---------------------------*/
/* ProcessInfo::setActualPID */
/*---------------------------*/
void ProcessInfo::setActualPID (int real_pid)
{
    actual_pid = real_pid;
}

/*------------------------*/
/* ProcessInfo::setStatus */
/*------------------------*/
void ProcessInfo::setStatus (int slaves_status)
{
    status = slaves_status;
}

/*-----------------------------*/
/* ProcessInfo::setProcRetCode */
/*-----------------------------*/
void ProcessInfo::setProcRetCode (int rc)
{
    procretcode = rc;
}


/*-----------------------*/
/* ProcessInfo::setInput */
/*-----------------------*/
void ProcessInfo::setInput (void * input, unsigned size)
{
    if (input_buff != NULL && size > 0)
         delete [] input_buff;

    insize = size;

    if (input == NULL)
         input_buff = NULL;
    else
         {
         input_buff = (char *) new char[size];
         memcpy(input_buff, input, size);
         }
}

/*---------------------------*/
/* ProcessInfo::setOutputPtr */
/*---------------------------*/
/*
    outptr is the address of the pointer that we want
    to save.  So!, all we have to do is copy the contents
    of &outptr to get the address!
*/
void ProcessInfo::setOutputPtr (Buffer * outptr)
{
    if ((void *)outptr == NULL)
         {
         size_of_output_ptr = 0;
         memset(output_ptr, 0, MAX_PTR_SIZE);
         }
    else
         {
         size_of_output_ptr = sizeof(outptr);
         memcpy(output_ptr, &outptr, sizeof(outptr));
         }

//  result_ptr = outptr;
}
```

```
/*
     clninfo.C
     Client System Information class.
     Brian Cox
*/

#include "clninfo.h"

/*-------------*/
/* Constructor */
/*-------------*/
ClientInfo::ClientInfo ()
{
     setCID(0);
     setStatus(0);
     setPort(0);
     setSocket(0);
     setHostName("");
     setClientName("");
}

/*------------------*/
/* Copy Constructor */
/*------------------*/
ClientInfo::ClientInfo (const ClientInfo &source)
{
     setCID(0);
     setStatus(0);
     setPort(0);
     setSocket(0);
     setHostName("");
     setClientName("");
     *this = source;
}

/*---------------------*/
/* Assignment Operator */
/*---------------------*/
ClientInfo& ClientInfo::operator = (const ClientInfo &source)
{
     // check if we are copying ourselves?
     if (this == &source)
          return *this;

     setCID(source.cid);
     setStatus(source.status);
     setClientName((char *)source.clientname);
     setHostName((char *)source.host_name);
     setPort(source.port);
     setSocket(sock);

     return *this;
}

/*-------------------*/
/* Equality Operator */
/*-------------------*/
int ClientInfo::operator == (const ClientInfo &source)
{
     return (getCID() == source.cid);
}

/*-------------*/
/* Save Object */
/*-------------*/
int ClientInfo::save (Buffer &b)
{
     // now copy each element of the class to the buffer
     SAVE(b, cid);
     SAVE(b, status);
     SAVE(b, clientname, (size_t)MAX_LEN_CLIENT_NAME);
     SAVE(b, host_name, (size_t)MAX_LEN_HOST_NAME);
     SAVE(b, port);
     SAVE(b, sock);

     return b.length;
}

/*-------------*/
/* Load Object */
/*-------------*/
Buffer ClientInfo::load (const Buffer& b)
{
     Buffer tmp = b;

     tmp = LOAD(tmp, cid);
     tmp = LOAD(tmp, status);
     tmp = LOAD(tmp, clientname, (size_t)MAX_LEN_CLIENT_NAME);
     tmp = LOAD(tmp, host_name, (size_t)MAX_LEN_HOST_NAME);
     tmp = LOAD(tmp, port);
     tmp = LOAD(tmp, sock);

     return tmp;
}

/*--------------------*/
/* ClientInfo::getCID */
/*--------------------*/
unsigned long ClientInfo::getCID ()
{
     return cid;
}

/*-----------------------*/
/* ClientInfo::getStatus */
/*-----------------------*/
int ClientInfo::getStatus ()
{
     return status;
}

/*---------------------------*/
/* ClientInfo::getClientName */
/*---------------------------*/
void ClientInfo::getClientName (char clnname[MAX_LEN_CLIENT_NAME])
{
     memcpy(clnname, clientname, MAX_LEN_CLIENT_NAME);
}
```

```
/*-------------------------*/
/* ClientInfo::getHostName */
/*-------------------------*/
void ClientInfo::getHostName (char hname[MAX_LEN_HOST_NAME])
{
     memcpy(hname, host_name, MAX_LEN_HOST_NAME);
}

/*---------------------*/
/* ClientInfo::getPort */
/*---------------------*/
int ClientInfo::getPort ()
{
     return port;
}

/*-----------------------*/
/* ClientInfo::getSocket */
/*-----------------------*/
int ClientInfo::getSocket ()
{
     return sock;
}

/*--------------------*/
/* ClientInfo::setCID */
/*--------------------*/
void ClientInfo::setCID (unsigned long id)
{
     cid = id;
}

/*-----------------------*/
/* ClientInfo::setStatus */
/*-----------------------*/
void ClientInfo::setStatus (int slaves_status)
{
     status = slaves_status;
}

/*---------------------------*/
/* ClientInfo::setClientName */
/*---------------------------*/
void ClientInfo::setClientName (char * clnname)
{
     strncpy(clientname, clnname, MAX_LEN_CLIENT_NAME);
     clientname[MAX_LEN_CLIENT_NAME-1] = '\0';
}

/*-------------------------*/
/* ClientInfo::setHostName */
/*-------------------------*/
void ClientInfo::setHostName (char * hname)
{
     strncpy(host_name, hname, MAX_LEN_HOST_NAME);
     host_name[MAX_LEN_HOST_NAME-1] = '\0';
}

/*---------------------*/
/* ClientInfo::setPort */
/*---------------------*/
void ClientInfo::setPort (int port_id)
{
     port = port_id;
}

/*-----------------------*/
/* ClientInfo::setSocket */
/*-----------------------*/
void ClientInfo::setSocket (int s)
{
     sock = s;
}
```

```
/*
     resinfo.C
     Result System Information class.
     Brian Cox
*/

#include "resinfo.h"

/*-------------*/
/* Constructor */
/*-------------*/
ResultInfo::ResultInfo ()
{
     setPID(0);
     result=NULL;       // setResult() would check for NULL
pointer!!
     resultsize=0;
}

/*------------------*/
/* Copy Constructor */
/*------------------*/
ResultInfo::ResultInfo (const ResultInfo &source)
{
     setPID(0);
     result=NULL;
     resultsize=0;
     *this = source;  // use assignment operator to copy rest of
object
}

/*------------*/
/* Destructor */
/*------------*/
ResultInfo::~ResultInfo ()
{
     if (result != NULL && resultsize > 0)
          delete [] result;
}

/*---------------------*/
/* Assignment Operator */
/*---------------------*/
ResultInfo& ResultInfo::operator = (const ResultInfo &source)
{
     // check if we are copying ourselves?
     if (this == &source)
          return *this;

     if (result != NULL && resultsize > 0)
          delete [] result;

     setPID(source.pid);
     setResult(source.result, source.resultsize);

     return *this;
}
/*-------------------*/
/* Equality Operator */
/*-------------------*/
int ResultInfo::operator == (const ResultInfo &source)
{
     return (pid == source.pid);
}

/*-------------*/
/* Save Object */
/*-------------*/
int ResultInfo::save (Buffer &b)
{
     // now copy each element of the class to the buffer
     SAVE(b, pid);
     SAVE(b, resultsize);
     if (resultsize > 0)
          SAVE(b, (char *)result, (size_t)resultsize);

     return b.length;
}

/*-------------*/
/* Load Object */
/*-------------*/
Buffer ResultInfo::load (const Buffer& b)
{
     Buffer tmp = b;

     tmp = LOAD(tmp, pid);
     tmp = LOAD(tmp, resultsize);
     if (resultsize > 0)
          {
          result = (void *)new char[resultsize];
          tmp = LOAD(tmp, (char *)result, (size_t)resultsize);
          }
     else
          result = NULL;

     return tmp;
}
/*--------------------*/
/* ResultInfo::getPID */
/*--------------------*/
unsigned long ResultInfo::getPID ()
{
     return pid;
}

/*-----------------------*/
/* ResultInfo::getResult */
/*-----------------------*/
//void ResultInfo::getResult (void *output)
void ResultInfo::getResult (Buffer *resbuf)
{
     Buffer b;

     b.data = new char[resultsize];
     b.length = resultsize;
     memcpy(b.data, result, resultsize);
     *resbuf = b;
}

/*--------------------*/
/* ResultInfo::setPID */
/*--------------------*/
void ResultInfo::setPID (unsigned long id)
{
     pid = id;
}

/*-----------------------*/
/* ResultInfo::setResult */
/*-----------------------*/
void ResultInfo::setResult (void * res, int size)
{
     if (result != NULL && resultsize > 0)
          delete [] result;

     if (res == NULL)
          {
          result = NULL;
          resultsize = 0;
          }
     else
          {
          resultsize = size;
          result = (void *)new char[size];
          memcpy(result, res, size);
          }
}
```

```
/*
     slvinfo.C
     Result System Information class.
     Brian Cox
*/

#include "slvinfo.h"

/*--------------*/
/* Constructor */
/*--------------*/
SlaveInfo::SlaveInfo ()
{
     setSID(0);
     setNumProcs(0);
     setStatus(0);
     setSocket(0);
     setPort(0);
     setHostName("");
}

/*-------------------*/
/* Copy Constructor */
/*-------------------*/
SlaveInfo::SlaveInfo (const SlaveInfo &source)
{
     *this = source;
}

/*----------------------*/
/* Assignment Operator */
/*----------------------*/
SlaveInfo& SlaveInfo::operator = (const SlaveInfo &source)
{
     // check if we are copying ourselves?
     if (this == &source)
          return *this;

     setSID(source.sid);
     setNumProcs(source.num_procs);
     setStatus(source.status);
     setHostName((char *)source.host_name);
     setPort(source.port);
     setSocket(source.sock);

     return *this;
}

/*--------------------*/
/* Equality Operator */
/*--------------------*/
int SlaveInfo::operator == (const SlaveInfo &source)
{
     return (sid == source.sid);
}

/*--------------*/
/* Save Object */
/*--------------*/
int SlaveInfo::save (Buffer &b)
{
     // now copy each element of the class to the buffer
     SAVE(b, sid);
     SAVE(b, num_procs);
     SAVE(b, status);
     SAVE(b, host_name, (size_t)MAX_LEN_HOST_NAME);
     SAVE(b, port);
     SAVE(b, sock);

     return b.length;
}

/*--------------*/
/* Load Object */
/*--------------*/
Buffer SlaveInfo::load (const Buffer& b)
{
     Buffer tmp = b;

     tmp = LOAD(tmp, sid);
     tmp = LOAD(tmp, num_procs);
     tmp = LOAD(tmp, status);
     tmp = LOAD(tmp, host_name, (size_t)MAX_LEN_HOST_NAME);
     tmp = LOAD(tmp, port);
     tmp = LOAD(tmp, sock);

     return tmp;
}

/*--------------------*/
/* SlaveInfo::getSID */
/*--------------------*/
unsigned long SlaveInfo::getSID ()
{
     return sid;
}

/*-------------------------*/
/* SlaveInfo::getNumProcs */
/*-------------------------*/
int SlaveInfo::getNumProcs ()
{
     return num_procs;
}

/*-----------------------*/
/* SlaveInfo::getStatus */
/*-----------------------*/
int SlaveInfo::getStatus ()
{
     return status;
}

/*-------------------------*/
/* SlaveInfo::getHostName */
/*-------------------------*/
void SlaveInfo::getHostName (char hname[MAX_LEN_HOST_NAME])
{
     memcpy(hname, host_name, MAX_LEN_HOST_NAME);
}

/*---------------------*/
/* SlaveInfo::getPort */
/*---------------------*/
int SlaveInfo::getPort ()
{
     return port;
}

/*-----------------------*/
/* SlaveInfo::getSocket */
/*-----------------------*/
int SlaveInfo::getSocket ()
{
     return sock;
}

/*--------------------*/
/* SlaveInfo::setSID */
/*--------------------*/
void SlaveInfo::setSID (unsigned long id)
{
     sid = id;
}

/*-------------------------*/
/* SlaveInfo::setNumProcs */
/*-------------------------*/
void SlaveInfo::setNumProcs (int num_curr_processes)
{
     num_procs = num_curr_processes;
}

/*-----------------------*/
/* SlaveInfo::setStatus */
/*-----------------------*/
void SlaveInfo::setStatus (int slaves_status)
{
     status = slaves_status;
}

/*-------------------------*/
/* SlaveInfo::setHostName */
/*-------------------------*/
void SlaveInfo::setHostName (char * hname)
{
     strncpy(host_name, hname, MAX_LEN_HOST_NAME);
     host_name[MAX_LEN_HOST_NAME-1] = '\0';
}

/*---------------------*/
/* SlaveInfo::setPort */
/*---------------------*/
void SlaveInfo::setPort (int port_id)
{
     port = port_id;
}

/*-----------------------*/
/* SlaveInfo::setSocket */
/*-----------------------*/
void SlaveInfo::setSocket (int s)
{
     sock = s;
}
```

```
/*
     waitinfo.C
     Wait System Information class.
     Brian Cox
*/

#include "waitinfo.h"

/*-------------*/
/* Constructor */
/*-------------*/
WaitInfo::WaitInfo ()
{
     setPID(0);
     setSock(0);
}

/*------------------*/
/* Copy Constructor */
/*------------------*/
WaitInfo::WaitInfo (const WaitInfo &source)
{
     *this = source;
}

/*---------------------*/
/* Assignment Operator */
/*---------------------*/
WaitInfo& WaitInfo::operator = (const WaitInfo &source)
{
     // check if we are copying ourselves?
     if (this == &source)
          return *this;

     setPID(source.pid);
     setSock(source.socket);
     return *this;
}

/*-------------------*/
/* Equality Operator */
/*-------------------*/
int WaitInfo::operator == (const WaitInfo &source)
{
     return (pid == source.pid);
}

/*-------------*/
/* Save Object */
/*-------------*/
int WaitInfo::save (Buffer &b)
{
     // now copy each element of the class to the buffer
     SAVE(b, pid);
     SAVE(b, socket);
     return b.length;
}

/*-------------*/
/* Load Object */
/*-------------*/
Buffer WaitInfo::load (const Buffer& b)
{
     Buffer tmp = b;
     tmp = LOAD(tmp. pid);
     tmp = LOAD(tmp. socket);
     return tmp;
}

/*-------------------*/
/* WaitInfo::getPID   */
/*-------------------*/
ulong WaitInfo::getPID ()
{
     return pid;
}

/*------------------*/
/* WaitInfo::getWait */
/*------------------*/
int WaitInfo::getSock ()
{
     return socket;
}

/*-----------------*/
/* WaitInfo::setPID */
/*-----------------*/
void WaitInfo::setPID (unsigned long id)
{
     pid = id;
}

/*------------------*/
/* WaitInfo::setSock */
/*------------------*/
void WaitInfo::setSock (int sock)
{
     socket = sock;
}
```

8.6 Miscellaneous

```
/*
     General functions for reporting system errors.
     Brian Cox.
     December 1993.
*/

#include "syserr.h"

void print_syserr(char * msg, unsigned short line, char * file)
{
     cerr << "(E) " << file << ", " << line << "; " << msg << ". " << flush;
     perror(NULL);
}

void print_syswarn(char * msg, unsigned short line, char * file)
{
     cerr << "(W) " << file << ", " << line << "; " << msg << ".\n" << flush;
}
```

```
/*
     buffer.C
     Brian Cox
     August 2nd 1993.

     Buffer class member functions.
*/

#include "buffer.h"

// Constructors
Buffer::Buffer()
{
     data = NULL;
     length = 0;
}

// Copy constructor
Buffer::Buffer (const Buffer &source)
{
     // make sure that we're not copying ourself
     if (this != &source)
          {
          //
          // Make sure that object is in a clean state
          // before going any further.
          //
          data = NULL;
          length = 0;
          *this = source;             // use assignment operator to copy elements)
          }
}

// Destructor
Buffer::~Buffer()
{
     empty();
}

// Empty buffer
void Buffer::empty()
{
     length = 0;
     if (data != NULL)
          delete [] data;
}

// Assignment operators
Buffer& Buffer::operator = (const Buffer &source)
{
     // check if we're copying ourself
     if (this == &source)
          return *this;

     if (data != NULL)  // free old data
          delete [] data;

     length = source.length;
     if (length == 0)
          data = NULL;
     else
          {
          data = new char[length];
          memcpy(data, source.data, length);
          }
     return *this; // pass this object back for multiple assignments
}

// Assignment operator
Buffer& Buffer::operator = (const char * source)
{
     if (data != NULL)  // free old data
          delete [] data;

     // catch NULL assignment => empty buffer
     if ( (source == NULL) || (strlen(source) == 0) )
          {
          length = 0;
          data = NULL;
          }
     else
          {
          length = strlen(source)+1;
          data = new char[length];
          memcpy(data, source, length);
          }
     return *this; // pass this object back for multiple assignments
}
```

8.7 Header Files

```
/*
     buffer.h
     Brian Cox
     August 2nd 1993.

     Class wrappers for some common data structures.
*/

#ifndef _BUFFER_H_
#define _BUFFER_H_

#include <string.h>
#include <memory.h>

class Buffer
{
public:
     int length;
     char * data;

     // constructors
     Buffer();
     -Buffer();
     Buffer(const Buffer &);                 // copy constructor
     Buffer(const char * &);

     // overloaded assignment operators
     Buffer& operator=(const Buffer &);
     Buffer& operator=(const char *);

     void empty();                           // empty the buffer!
};

#endif
```

```
/*
     clientrm.h
     Client System Tables
     Brian Cox

     Only one instance of each resource manager is required
     in a Server therefore all members can be static!
*/

#ifndef _CLIENTRM_H_
#define _CLIENTRM_H_

#include "distproc.h"
#include "clninfo.h"
#include "systab.h"

class ClientResMgr : public SystemTable<ClientInfo>
{
public:
     Attach (ClientInfo &cinfo);
     Detach (ulong cid);

     Find (ulong cid, ClientInfo &cinfo);

protected:
     static unsigned long client_id;
};

#endif /* ifndef _CLIENTRM_H_ */
```

```
/*
     clnhndls.h
     Brian Cox
     September 1993

     Distributed Processing client - client message handler
                                     function headers.
*/

#ifndef _CLNHNDLS_H_
#define _CLNHNDLS_H_

#include 'distproc.h"
#include 'buffer.h"

typedef unsigned long ulong;

/*------------------------*/
/* DistProc Client class. */
/*------------------------*/
class DPClient
{
public:
     DPClient(char *servername);
     ~DPClient();

     ulong run (char *filename, Buffer &inbuf, Buffer *resbuf);
     int  kill (ulong pid);
     int  wait (ulong pid);
     //int  waitany (ulong &pid);

     int Registered();

protected:
     int MakeRequest (Buffer request, Buffer &response);
     int ConnectToServer (char * dpsrvhostname);
     int DisconnectFromServer ();

private:
     char * DPServerHostName;     // name of DistProc Server's host
     ulong  ClientID;             // id # for this client
     int    registered;           // did we register sucessfully?
     static char *ClientVer;      // client requester version
number

     int    sock;                 // socket to DP Server
};

#endif /* ifndef _CLNHNDLS_H_ */
```

```
/*
     clninfo.h
     Client Information class.
     Brian Cox
*/

#ifndef _CLNINFO_H_
#define _CLNINFO_H_

#include <stdio.h>
#include "distproc.h"
#include "saveobj.h"
#include "buffer.h"

/*-----------------------------------------*/
/* record structure of Client system table. */
/*-----------------------------------------*/
class ClientInfo
{
public:
     ClientInfo ();
     ClientInfo (const ClientInfo &source);
     ClientInfo& operator = (const ClientInfo &source);
     int operator == (const ClientInfo &source);

     int    save (Buffer &b);
     Buffer load (const Buffer& b);

     unsigned long getCID ();
     int  getStatus ();
     void getClientName (char clnname[MAX_LEN_CLIENT_NAME]);
     void getHostName (char clnname[MAX_LEN_HOST_NAME]);
     int  getPort ();
     int getSocket ();

     void setCID (unsigned long id);
     void setStatus (int slaves_status);
     void setClientName (char * clnname);
     void setHostName (char * hname);
     void setPort (int port);
     void setSocket (int s);

protected:
     unsigned long cid; /* Clients system id */
     int  status;        /* current status of the Client: WAITING
etc.*/

     char clientname[MAX_LEN_CLIENT_NAME];

     /* network dependent bit */
     int  port;           /* port on which to contact the Client */
     char host_name[MAX_LEN_HOST_NAME]; /* Clients internet
address */
     int  sock;          /* socket on which client is attached */
};


#endif /* ifndef _CLNINFO_H_ */
```

```
/*
     distproc.h
     Brian Cox
     9th September 1993

     Distributed Processing System - various defines used
throughout
     the system.
*/

#ifndef _DISTPROC_H_
#define _DISTPROC_H_

#include <limits.h>
#include <sys/param.h>
#include "syserr.h"

/*
** maximum size of various system names.
*/
#define MAX_LEN_CLIENT_NAME _POSIX_NAME_MAX
#define MAX_LEN_HOST_NAME   MAXHOSTNAMELEN
#define MAX_LEN_IMAGENAME   MAXPATHLEN
//#define MAX_LEN_CLIENT_NAME       128
//#define MAX_LEN_HOST_NAME 128
//#define MAX_LEN_IMAGENAME 128

/*
** network specific constants.
*/
#define SERVER_TCP_PORT                    6543
#define SLAVE_TCP_PORT                     6544
#define SLAVE_REMPROC_TCP_PORT             6545
#define SLAVE_REMPROC_TCP_PORT_STRING     "6545"

/*
** client should figure out its own port number.
*/
#define CLIENT_TCP_PORT     6546
#define SERVER_HOST_ADDR    "popeye"

/*
** Various type of requesters.
*/
#define SLAVE_REQUESTER       1
#define CLIENT_REQUESTER      2
#define SERVER_REQUESTER      3

/*
** system status variables.
*/
#define REQ_UNAVAILABLE           100
#define REQ_DEAD                  101
#define REQ_RUNNING               102

#define SLAVE_UNAVAILABLE   REQ_UNAVAILABLE
#define SLAVE_DEAD          REQ_DEAD
#define SLAVE_RUNNING       REQ_RUNNING

#define CLIENT_UNAVAILABLE  REQ_UNAVAILABLE
#define CLIENT_DEAD         REQ_DEAD
#define CLIENT_RUNNING      REQ_RUNNING

#define PROCESS_ALLOCATED   200
#define PROCESS_RUNNING     201
#define PROCESS_FINISHED    202
#define PROCESS_ABORTED     203

/*
** if a distproc pid is zero then it's bad
*/
#define INVALID_DPID          0

#endif
```

```
/*
     dpclient.h
     Brian Cox
     September 1993

     Distributed Processing Client header file.
*/

#ifndef _DPCLIENT_H_
#define _DPCLIENT_H_

#include "distproc.h"       /* general purpose defines. */
#include "clnhndls.h"
#include "buffer.h"
#include "saveobj.h"

typedef unsigned long DPID;

#endif  /* #ifndef _DPCLIENT_H_  */
```

```
/*
   linklist.h
   A template for a linked list.
   B.Cox, 12/1994.
*/


#ifndef _LINKLIST_H
#define _LINKLIST_H

//----------------------
// the linked list entry
//----------------------
template <class T>
class ListEntry    {
      T thisentry;
      ListEntry<T> *nextentry;
      ListEntry<T> *preventry;
      ListEntry(T& entry);
      friend class LinkedList<T>;
};


// ---- construct a linked list entry
template <class T>
ListEntry<T>::ListEntry(T &entry)
{
      thisentry = entry;
      nextentry = NULL;
      preventry = NULL;
}


//----------------
// the linked list
//----------------
template <class T>
class LinkedList    {
     // --- the listhead
     ListEntry<T> *firstentry;
     ListEntry<T> *lastentry;
     ListEntry<T> *iterator;
     void RemoveEntry(ListEntry<T> *lentry);
     void InsertEntry(T& entry, ListEntry<T> *lentry);
public:
     LinkedList();
     ~LinkedList();
     void AppendEntry(T& entry);
     void RemoveEntry(int pos = -1);
     void InsertEntry(T&entry, int pos = -1);
     T *FindEntry(int pos);
     T *CurrentEntry();
     T *FirstEntry();
     T *LastEntry();
     T *NextEntry();
     T *PrevEntry();
};


// ---- construct a linked list
template <class T>
LinkedList<T>::LinkedList()
{
     iterator = NULL;
     firstentry = NULL;
     lastentry = NULL;
}


// ---- destroy a linked list
template <class T>
LinkedList<T>::~LinkedList()
{
     while (firstentry)
           RemoveEntry(firstentry);
}


// ---- append an entry to the linked list
template <class T>
void LinkedList<T>::AppendEntry(T& entry)
{
     ListEntry<T> *newentry = new ListEntry<T>(entry);
     newentry->preventry = lastentry;
     if (lastentry)
          lastentry->nextentry = newentry;
     if (firstentry == NULL)
          firstentry = newentry;
     lastentry = newentry;
}


// ---- remove an entry from the linked list
template <class T>
void LinkedList<T>::RemoveEntry(ListEntry<T> *lentry)
{
     if (lentry == NULL)
          return;
     if (lentry == iterator)
          iterator = lentry->preventry;
     // ---- repair any break made by this removal
     if (lentry->nextentry)
          lentry->nextentry->preventry = lentry->preventry;
     if (lentry->preventry)
          lentry->preventry->nextentry = lentry->nextentry;
     // --- maintain listhead if this is last and/or first
     if (lentry == lastentry)
          lastentry = lentry->preventry;
     if (lentry == firstentry)
          firstentry = lentry->nextentry;
     delete lentry;
}


// ---- insert an entry into the linked list
template <class T>
void LinkedList<T>::InsertEntry(T& entry, ListEntry<T> *lentry)
{
     ListEntry<T> *newentry = new ListEntry<T>(entry);
     newentry->nextentry = lentry;
     if (lentry)    {
          newentry->preventry = lentry->preventry;
          lentry->preventry = newentry;
     }
     if (newentry->preventry)
          newentry->preventry->nextentry = newentry;
    if (lentry == firstentry)
          firstentry = newentry;
}


// ---- remove an entry from the linked list
template <class T>
void LinkedList<T>::RemoveEntry(int pos)
{
      FindEntry(pos);
      RemoveEntry(iterator);
}


// ---- insert an entry into the linked list
template <class T>
void LinkedList<T>::InsertEntry(T& entry, int pos)
{
      FindEntry(pos);
      InsertEntry(entry, iterator);
}


// ---- return the current linked list entry
template <class T>
T *LinkedList<T>::CurrentEntry()
{
    return iterator ? &(iterator->thisentry) : NULL;
}


// ---- return a specific linked list entry
template <class T>
T *LinkedList<T>::FindEntry(int pos)
{
      if (pos != -1)    {
            iterator = firstentry;
            if (iterator)    {
                    while (pos--)
                          iterator = iterator->nextentry;
            }
      }
      return CurrentEntry();
}


// ---- return the first entry in the linked list
template <class T>
T *LinkedList<T>::FirstEntry()
{
      iterator = firstentry;
      return CurrentEntry();
}


// ---- return the last entry in the linked list
template <class T>
T *LinkedList<T>::LastEntry()
{
      iterator = lastentry;
      return CurrentEntry();
}


// ---- return the next entry in the linked list
template <class T>
T *LinkedList<T>::NextEntry()
{
      if (iterator == NULL)
            iterator = firstentry;
      else
            iterator = iterator->nextentry;
      return CurrentEntry();
}


// ---- return the previous entry in the linked list
template <class T>
T *LinkedList<T>::PrevEntry()
{
      if (iterator == NULL)
            iterator = lastentry;
      else
            iterator = iterator->preventry;
      return CurrentEntry();
}


#endif
```

```
/*
     messages.h
     Brian Cox.
     August 1st 1993.

     All the system message manipulator classes are in this file.
*/

#ifndef _MESSAGES_H_
#define _MESSAGES_H_

#include "prcinfo.h"
#include "resinfo.h"
#include "slvinfo.h"
#include "clninfo.h"
#include "buffer.h"
#include "msgdefs.h"
//#include <mem.h>

/*---------*/
/* BaseMsg */
/*---------*/

class BaseMsg
{
public:

     BaseMsg ();                                  // constructor
     BaseMsg (Buffer b);                          // re-constructor

     unsigned long getSystemID ()  { return system_id; };
     Service       getService () { return sv; };
     void          setSystemID (unsigned long id) { system_id =
id; };

     /* persistance member functions */
     int save (Buffer &b);           // save this message to a
buffer
     Buffer load (const Buffer& b); // create a new message from a
buffer

     int retcode;

protected:
     int              length;         // size of message, not really
used
     Service          sv;                         // type of service
     unsigned long system_id;        // client or slave id #
     int              req_type;                   // requester type
- server, slave or client.

     void setService (Service service) { sv = service; };
};


/*--------------------------------------------*/
/* ServerShutDownMsg : virtual public BaseMsg */
/*--------------------------------------------*/

class ServerShutDownMsg : virtual public BaseMsg
{
public:
     ServerShutDownMsg ();
     ServerShutDownMsg (Buffer b);

     int    save (Buffer &b);
     Buffer load (const Buffer& b);
};


/*----------------------------------*/
/* DOCMsg : virtual public BaseMsg */
/*----------------------------------*/

class DOCMsg : virtual public BaseMsg
{
public:
     DOCMsg ();
     DOCMsg (Buffer b);

     int    save (Buffer &b);
     Buffer load (const Buffer& b);

     unsigned long sid;
     int           realpid;
};


/*--------------------------------------------*/
/* ClientConnectMsg : virtual public BaseMsg */
/*--------------------------------------------*/

class ClientConnectMsg : virtual public BaseMsg
{
public:
     ClientConnectMsg ();
     ClientConnectMsg (Buffer b);

     int    save (Buffer &b);
     Buffer load (const Buffer& b);

     ClientInfo cinfo;               // client information block
};


/*-----------------------------------------------*/
/* ClientDisconnectMsg : virtual public BaseMsg */
/*-----------------------------------------------*/

class ClientDisconnectMsg : virtual public BaseMsg
{
public:
     ClientDisconnectMsg ();
     ClientDisconnectMsg (Buffer b);

     int    save (Buffer &b);
     Buffer load (const Buffer& b);
};
```

```
/*----------------------------------------*/
/* RunProcessMsg : virtual public BaseMsg */
/*----------------------------------------*/

class RunProcessMsg : virtual public BaseMsg
{
public:
     RunProcessMsg ();
     RunProcessMsg (Buffer b);

     int    save (Buffer &b);
     Buffer load (const Buffer& b);

     ProcessInfo pinfo;
};


/*-----------------------------------------*/
/* WaitProcessMsg : virtual public BaseMsg */
/*-----------------------------------------*/

class WaitProcessMsg : virtual public BaseMsg
{
public:
     WaitProcessMsg ();
     WaitProcessMsg (Buffer b);

     int    save (Buffer &b);
     Buffer load (const Buffer& b);

     ProcessInfo pinfo;      /* various process information */
     ResultInfo    rinfo;      /* result information */
};


/*-----------------------------------------*/
/* KillProcessMsg : virtual public BaseMsg */
/*-----------------------------------------*/

class KillProcessMsg : virtual public BaseMsg
{
public:
     KillProcessMsg ();
     KillProcessMsg (Buffer b);

     int    save (Buffer &b);
     Buffer load (const Buffer& b);

     ProcessInfo pinfo;
};


/*-------------------------------------*/
/* ResultMsg : virtual public BaseMsg */
/*-------------------------------------*/

class ResultMsg : virtual public BaseMsg
{
public:
     ResultMsg();
     ResultMsg(Buffer b);

     int    save (Buffer &b);
     Buffer load (const Buffer& b);

     ResultInfo rinfo;
};


/*------------------------------------------*/
/* SlaveConnectMsg : virtual public BaseMsg */
/*------------------------------------------*/

class SlaveConnectMsg : virtual public BaseMsg
{
public:
     SlaveConnectMsg();
     SlaveConnectMsg(Buffer b);

     int    save (Buffer &b);
     Buffer load (const Buffer& b);

     SlaveInfo sinfo;
};


/*---------------------------------------------*/
/* SlaveDisconnectMsg : virtual public BaseMsg */
/*---------------------------------------------*/

class SlaveDisconnectMsg : virtual public BaseMsg
{
public:
     SlaveDisconnectMsg();
     SlaveDisconnectMsg(Buffer b);

     int    save (Buffer &b);
     Buffer load (const Buffer& b);
};


/*------------------------------------------*/
/* QueryStatusMsg : virtual public BaseMsg */
/*------------------------------------------*/

class QueryStatusMsg : virtual public BaseMsg
{
public:
     QueryStatusMsg();
     QueryStatusMsg(Buffer b);

     int    save (Buffer &b);
     Buffer load (const Buffer& b);
};

#endif
```

```
/*
     msgdefs.h
     Brian Cox.
     August 1st 1993.

     All the system message data types are defined here.
*/

#ifndef _MSGDEFS_H_
#define _MSGDEFS_H_

typedef int NetAddr;
typedef int MsgID;

enum _service
{
     no_service = 0,
     assoc_proc,
     run_proc,
     wait_proc,
     wait_any_proc,
     kill_proc,
     client_connect,
     client_shutdown,
     server_shutdown,
     slave_connect,
     slave_shutdown,
     result,
     query_status,
     death_of_child
};
typedef enum _service Service;


int const MAX_PNAME_LEN = 20;
int const MAX_FNAME_LEN = 128;

typedef char ProcNm[MAX_PNAME_LEN];
typedef char FileNm[MAX_FNAME_LEN];

#endif
```

```
/*
     prcinfo.h
     Remote Process Information class.
     Brian Cox.
*/

#ifndef _PRCINFO_H_
#define _PRCINFO_H_

#include "distproc.h"
#include "saveobj.h"
#include "buffer.h"
//#include <mem.h>

#define MAX_PTR_SIZE    10

/*--------------------------------------------*/
/* record structure for Process system table. */
/*--------------------------------------------*/
class ProcessInfo
{
public:
     ProcessInfo ();
     ProcessInfo (const ProcessInfo &source);
     ~ProcessInfo ();

     ProcessInfo& operator =  (const ProcessInfo &source);
     int          operator == (const ProcessInfo &source);

     int    save (Buffer &b);
     Buffer load (const Buffer& b);

     unsigned long getPID ();
     unsigned long getCID ();
     unsigned long getSID ();
     void getClientName (char clnname[MAX_LEN_CLIENT_NAME]);
     void getFileName (char fname[MAX_LEN_IMAGENAME]);
     int  getActualPID ();
     int  getStatus ();
     int  getProcRetCode ();
     void getInput (void * &input, unsigned &size);
     void getOutputPtr (Buffer* outptr);

     void setPID (unsigned long procid);
     void setCID (unsigned long clientid);
     void setSID (unsigned long slaveid);
     void setClientName (char * clnname);
     void setFileName (char * fname);
     void setActualPID (int real_pid);
     void setStatus (int slaves_status);
     void setProcRetCode (int rc);
     void setInput (void * input, unsigned size);
     void setOutputPtr (Buffer * outptr);

protected:
     unsigned long pid;          // id for this process
     unsigned long cid;          // id of Client which started
this process
     unsigned long sid;          // id of Slave which is running
the process
     int          actual_pid;   // actual pid of process on slave
     int          status;       // status of process; running,
finished etc.
     int          procretcode;     // return code from
Slave/Process
     char         clientname[MAX_LEN_CLIENT_NAME];
     char         filename[MAX_LEN_IMAGENAME];

     int          size_of_output_ptr;
     char         output_ptr[MAX_PTR_SIZE];// buffer to hold the
output pointer
//   Buffer*      result_ptr;

     unsigned     insize;       // size of the input
     void *       input_buff;   // pointer to the input
};

#endif /* #ifndef _PRCINFO_H_ */
```

```
/*
    procrm.h
    Process Resource Management.
    Brian Cox.

    Only one instance of each resource manager is required
    in a Server therefore all members can be static!
*/

#ifndef _PROCRM_H_
#define _PROCRM_H_

#include "distproc.h"
#include "prcinfo.h"
#include "systab.h"

class ProcessResMgr : public SystemTable<ProcessInfo>
{
public:
    ProcessResMgr();

    Alloc (ProcessInfo &pinfo);
    Dealloc (unsigned long pid);

    Find (ulong pid, ProcessInfo &pinfo);

    unsigned int NumProcesses;

protected:
    // public data members.  needed by srvhndls.
    static unsigned long process_id;
};

#endif /* #ifndef _PROCESS_H_  */
```

```
/*
    Brian Cox.
    August 1993.

    Queue template class and member functions.

    Note:
    If this class is used in a multi-threaded operating system
    then the Queue primitives get and put must be protected
    against deadlock etc. This is best done by defining a new
    wrapper class which controls access to the base Queue class.
*/

#ifndef _QUEUE_H_
#define _QUEUE_H_

int const ERR_QUEUE_FULL = 1;
int const ERR_QUEUE_EMPTY = 2;
#define DEFAULT_Q_SIZE 500

template <class T> class Queue
{
public:
    Queue();
    ~Queue();

    put(T);
    get(T&);

    peek(T&);
    getItemCount();

protected:
    int QSize;
    int itemCount,          // number of elements in the queue
        Qfront,                     // the element at the front of
the queue
        Qend;                       // the back of the queue
    T q[DEFAULT_Q_SIZE];
};


// -------------
// Constructor
// -------------
template <class T>
Queue<T>::Queue()
{
    QSize = DEFAULT_Q_SIZE;
    Qfront = 0;
    Qend = 0;
    itemCount = 0;
}

// Destroy items on queue.
template <class T>
Queue<T>::~Queue()
{
}

template <class T>
Queue<T>::put(T item)
{
    if (getItemCount() < QSize)     // check if the queue is full?
        {
        q[Qend] = item;             // add item to queue
        itemCount++;                // increment item count
        if (++Qend == QSize)        // are we at the end of the
array?
            Qend = 0;               // wrap around to start of
array again
        return 0;
        }
    else
        return ERR_QUEUE_FULL;
    // queue is full
}

template <class T>
Queue<T>::get(T &item)
{
    if (getItemCount() > 0)
        {
        item = q[Qfront];
        itemCount--;
        if (++Qfront == QSize)
            Qfront = 0;             // wrap around to start of
array again
        return 0;                   // everything's OK
        }
    else
        return ERR_QUEUE_EMPTY;
}

template <class T>
Queue<T>::peek(T &item)
{
    if (getItemCount() > 0)
        {
        item = q[Qfront];
        return 0;
        }
    else
        return ERR_QUEUE_EMPTY;
}

template <class T>
Queue<T>::getItemCount()
{
    return itemCount;
}

#endif  // _QUEUE_H_
```

```
/*
     resinfo.h
     Result System Information class.
     Brian Cox
*/

#ifndef _RESINFO_H_
#define _RESINFO_H_

#include "distproc.h"
#include "buffer.h"
#include "saveobj.h"

/*-----------------------------------------*/
/* record structure of Result system table. */
/*-----------------------------------------*/
class ResultInfo
{
public:
     ResultInfo ();
     ResultInfo (const ResultInfo &source);
     ~ResultInfo ();

     ResultInfo& operator =  (const ResultInfo &source);
     int         operator == (const ResultInfo &source);

     int    save (Buffer &b);
     Buffer load (const Buffer& b);

     unsigned long getPID ();
//   void           getResult (void *output);
     void           getResult (Buffer *resbuf);

     void setPID (unsigned long id);
     void setResult (void * res, int size);

protected:
     unsigned long  pid;           // id of RemProc that generated
the result
     int             resultsize;   // size of the result
     void *         result;        // the result
};

#endif  /* #ifndef _RESINFO_H_ */
```

```
/*
     resmgr.h
     Brian Cox
     March 1994.
*/

#ifndef _RESMGR_H_
#define _RESMGR_H_

#include "distproc.h"
#include "systab.h"

template <class INFO>
class ResourceManager
{
public:
     ResourceManager();
     ~ResourceManager();

     Find (unsigned long id, INFO &info);
     FindFirst (INFO &info);
     FindNext (INFO &info);

protected:
     virtual SystemTable<unsigned long> systab;
};

#endif  /* #ifndef */
```

```
/*
     resultrm.h
     Result System Tables
     Brian Cox
*/

#ifndef _RESULTRM_H_
#define _RESULTRM_H_

#include "distproc.h"
#include "resinfo.h"
#include "systab.h"

class ResultResMgr : public SystemTable<ResultInfo>
{
public:
     Add (ResultInfo rinfo);
     Delete (ulong pid);

     Find (ulong pid, ResultInfo &rinfo);
};

#endif  /* #ifndef _RESULTS_H_ */
```

```
/*
     rpargio.h
     Brian Cox
     Remote Process ARGument Input/Output.
*/

#ifndef _RPARGIO_H_
#define _RPARGIO_H_

#include "syserr.h"
#include "buffer.h"
#include "saveobj.h"

/*
** Global variables.
*/
extern char * HostNameForSlave;
extern unsigned long RemProcID;
extern unsigned long SlaveID;

/*
** Remote Process ARGument Input/Output functions.
*/
extern int GetInputParameters (Buffer &inbuf);
extern int ReturnResults (Buffer &resbuf);

#endif
```

```
/*
     saveobj.h
     Brian Cox.
     August 2nd 1993.

     A template based approach to saving objects.  No .cpp file to
go
     with this header - only templates defined here!

     The following data types can be saved:

          (1)  all fundamental data types in C++
          (2)  arrays and memory blocks of any fundamental data
type
          (3)  the Buffer class

     To handle additional data types/structures simply over-ride
     the SAVE() and LOAD() functions.
*/

#ifndef _SAVEOBJ_H_
#define _SAVEOBJ_H_

#include "buffer.h"
#include <string.h>
#include <stdlib.h>

// Integral data types - int, char, double, float etc.
template <class T> void SAVE (Buffer &b, T elem)
{
     size_t elemsize;

     elemsize = sizeof(T);
     if (elemsize > 0)
          {
                                                       // fixup
buffer size
          b.data = (char *)realloc((void *)b.data,
b.length+elemsize);
          memcpy(b.data+b.length, &elem, elemsize); // copy
element to buffer
          b.length += elemsize;                    // update
buffer size
          }
}

// Pointer to block of memory - Array or pointer. Won't take 'void
*'.
template <class T> void SAVE(Buffer &b, T * elem, size_t
num_elems)
{
     size_t elemsize = 0, total_size = 0;

     if ( (elem != NULL) && (num_elems > 0) )
          {
          elemsize = sizeof(T);
          total_size = elemsize * num_elems;

          b.data = (char *)realloc((void *)b.data,
b.length+total_size);
          memcpy(b.data+b.length, elem, total_size); // copy
element to buffer
          b.length += total_size;                   // update
buffer size
          }
}

// Buffer object
static void SAVE(Buffer &b, Buffer elem)
{
     SAVE(b, elem.length);
     SAVE(b, elem.data, (size_t)elem.length);
}

/*================================================================*/
/*      Load template functions - reconstruct an object          */
/*================================================================*/

// Integral data types - int, char, double, float etc.
template <class T> Buffer LOAD (const Buffer &b, T &elem)
{
     Buffer tmp;
     size_t elemsize;

     elemsize = sizeof(T);
     if (b.length < elemsize)
          return b;
     else
          {
          memcpy(&elem, b.data, elemsize);
          tmp.length = b.length - elemsize;               // size
of buffer left
          if (tmp.length > 0)
               {
               tmp.data = new char[tmp.length];           // next
element in buffer
               memcpy(tmp.data, b.data+elemsize, tmp.length);
               }
          }

     return tmp;
}

// Pointer to block of memory - Array or pointer. Won't take 'void
*'.
// elem is referenced only to allow elem = NULL in case num_elems
is 0.
// I had to remove 'T * &elem' as it wouldn't work with pointers.
template <class T> Buffer LOAD (const Buffer &b, T * elem, size_t
num_elems)
{
     Buffer tmp;
     size_t elemsize, total_size;

     elemsize = sizeof(T);
     total_size = elemsize * num_elems;

     //
     // make sure there's enough data left to copy
     //
     if ((num_elems == 0) || (b.length < total_size))
          return b;
     else
          {
          memcpy(elem, b.data, total_size);            // copy data
to elem
          tmp.length = b.length - total_size;          // size of
buffer left
          if (tmp.length > 0)
               {
               tmp.data = new char[tmp.length];         // next
element in buffer
               memcpy(tmp.data, b.data+total_size, tmp.length);
               }
          }

     return tmp;
}

#endif
```

```
#endif
```

```
/*
     slaverm.h
     Slave Resource Management class.
     Brian Cox.
*/

#ifndef _SLAVERM_H_
#define _SLAVERM_H_

#include "distproc.h"
#include "slvinfo.h"
#include "systab.h"

class SlaveResMgr : public SystemTable<SlaveInfo>
{
public:
     SlaveResMgr();

     Attach (SlaveInfo &sinfo);
     Detach (ulong sid);

     Alloc (ulong &sid);
     Dealloc (ulong sid);

     Find (ulong sid, SlaveInfo &sinfo);

     unsigned int NumSlaves;

protected:
     static ulong slave_id;
     ulong curr_slave;
};

#endif  /* #ifndef */
```

```
/*
     slvhndls.h
     Brian Cox

     Slave Message Handler functions declared in slhndlrs.C.
*/

#ifndef _SLVHNDLS_H_
#define _SLVHNDLS_H_


#include <sys/types.h>
#include <sys/time.h>
#include <sys/select.h>
#include "distproc.h"
#include "buffer.h"
#include "tcp.h"


/*----------------------*/
/* DistProc Slave class */
/*----------------------*/
class DPSlave
{
public:
     DPSlave(char *servername);
     ~DPSlave();

     WaitRequest(int &sock);
     DispatchRequest(int sock);

     int Registered();

     static const char * const Version;  // slave version number
     int SlaveOK;

protected:
     // server -> slave messages
     int RunProcess (int s, const Buffer &b);
     int KillProcess (int s, const Buffer &b);
     int ServerShutDown (int s, const Buffer &b);
     int CheckSlave (int s, const Buffer &b);

     // public data members
     static unsigned long SlaveID;     // slaves system id
     int SlaveRemProcSock;             // Slaves RemProc socket
     static int slavesock;

     int nfds;
     fd_set afds, rfds;
     struct timeval tout;

private:
     static void DeathOfChildHandler (int sig);

     static int MakeRequest (const Buffer &request, Buffer
&response);
     int FulfillRequest (int s, const Buffer &response);

     int ConnectToServer (char * srvname);
     int DisconnectFromServer ();

     int SpawnProcess (char *clientname, char *filename,
                       void *input, unsigned int insize,
                       unsigned long pid, int &actual_pid);
     int  KillChild (int pid);
     void KillAllChildren ();

     static char * DPServerHostName;   // name of DistProc
Server's host
     char    SlaveHostName[MAXHOSTNAMELEN];
     int     registered;
};
```

```
/*
     slvinfo.h
     Slave Information class.
     Brian Cox.
*/

#ifndef _SLVINFO_H_
#define _SLVINFO_H_

#include "distproc.h"
#include "saveobj.h"
#include "buffer.h"

/*---------------------------------------------*/
/* record structure for the Slave system table. */
/*---------------------------------------------*/
class SlaveInfo
{
public:
     SlaveInfo ();
     SlaveInfo (const SlaveInfo &source);
     SlaveInfo& operator = (const SlaveInfo &source);
     int operator == (const SlaveInfo &source);

     int    save (Buffer &b);
     Buffer load (const Buffer& b);

     unsigned long getSID ();
     int  getNumProcs ();
     int  getStatus ();
     void getHostName (char hname[MAX_LEN_HOST_NAME]);
     int  getPort ();
     int getSocket ();

     void setSID (unsigned long id);
     void setNumProcs (int num_curr_processes);
     void setStatus (int slaves_status);
     void setHostName (char * hname);
     void setPort (int port);
     void setSocket (int s);

protected:
     unsigned long sid; /* id of a Slave */
     int  num_procs;          /* number of processes running on the
Slave */
     int  status;             /* current status of the Slave */

     /* network dependent bit */
     char host_name[MAX_LEN_HOST_NAME];          /* Slaves internet
name */
     int  port;
     /* Slaves contact port */
     int  sock;                              /* Socket on which slave
is attached */
};

#endif  /* #ifndef */
```

```
/*
     srvhndls.h
     Brian Cox
     September 1993

     Distributed Processing Server class.
*/

#ifndef _SRVHNDLS_H_
#define _SRVHNDLS_H_


#include <stdio.h>
#include "srvprim.h"

class DPServer : public DPServerPrimitives
{
public:
     DPServer();
     ~DPServer();

     WaitRequest(int &sock);
     DispatchRequest(int sock);

protected:
     ClientAttachHnd (int sock, Buffer &b);
     ClientDetachHnd (int sock, Buffer &b);

     RunProcessHnd (int sock, Buffer &b);
     WaitProcessHnd (int sock, Buffer &b);
     KillRemProcHnd (int sock, Buffer &b);

     SlaveAttachHnd (int sock, Buffer &b);
     SlaveDetachHnd (int sock, Buffer &b);
     ResultFromRemProcHnd (int sock, Buffer &b);

     DeathOfChildHnd (int sock, Buffer &b);
};

#endif /* #ifndef  _SRVHNDLS_H_ */
```

```
/*
     srvprim.h
     Brian Cox
     March 1994

     Distributed Processing Server primitives class.
*/

#ifndef _SRVPRIM_H_
#define _SRVPRIM_H_

#include <stdio.h>
#include <sys/types.h>
#include <sys/time.h>
#include <sys/select.h>
#include "distproc.h"
#include "messages.h"
#include "tcp.h"
#include "clientrm.h"
#include "slaverm.h"
#include "procrm.h"
#include "resultrm.h"
#include "waitrm.h"
#include "queue.h"

class ErrInfo
{
public:
     int sock;
     unsigned long id;
     int reqtype;
};

class DPServerPrimitives
{
public:
     DPServerPrimitives();
     ~DPServerPrimitives();

     static const char * const Version;
     int ServerOK;

protected:
     int serversock;
     int nfds;
     fd_set afds, rfds, wfds;
     struct timeval tout;
                                                                   //
Communications methods...
     MakeRequest (unsigned long sid, Buffer request, Buffer
&reply);
     FulfillRequest (int sock, Buffer response);
                                                                   //
client primitives...
     ClientAttach (ClientConnectMsg &msg);
     ClientDetach (ClientDisconnectMsg &msg);
                                                                   //
slave primitives...
     SlaveAttach (SlaveConnectMsg &msg);
     SlaveDetach (SlaveDisconnectMsg &msg);
     RemProcReturningResult (ResultMsg &msg);
                                                                   //
process primitives...
     RunProcess (RunProcessMsg &msg);
     KillProcess (unsigned long pid);
     WaitProcess (WaitProcessMsg &msg, int sock);

     DeathOfChildRemProc (DOCMsg &msg);

     //
     // Error handling stuff.
     //
     Queue<ErrInfo> ErrQ;
     int  ErrorOnSocket (int sock);
     void ErrorOnSocket(int sock, unsigned long id, int reqtype);
     void ClearUpErrors();

private:
                                        // Resource manager
objects...
     ResultResMgr  resrm;
     SlaveResMgr   slaverm;
     ProcessResMgr procrm;
     ClientResMgr  clientrm;
     WaitResMgr    waitrm;

     void InformClientsOfShutdown();
     int InformSlavesOfShutdown();
};

#endif /* #ifndef  _SRVPRIM_H_ */




/*
     System error reporting functions.
     Brian Cox
*/

#ifndef _SYSERR_H_
#define _SYSERR_H_

#include <stdio.h>
#include <iostream.h>

extern int errno;

/* true & false. */
#ifndef FALSE
#define FALSE      0
#endif

#ifndef TRUE
#define TRUE       (!FALSE)
#endif

/* some constants and flags used in the program.*/
#define FAILED  -1
```

```
#define GOOD     0
#define WARNING     1

extern void print_syswarn (char * msg, unsigned short line, char *
file);
extern void print_syserr (char * msg, unsigned short line, char *
file);

/* macro to display debugging info */
#ifdef debug_on
#define debug(str) cerr << endl << str
#else
#define debug(str)
#endif

#define syswarn(msg) print_syswarn(msg, __LINE__, __FILE__)
#define syserr(msg) print_syserr(msg, __LINE__, __FILE__)

#endif
```

```
/*
     systab.h
     Basic System Table class.
     Brian Cox.
*/

#ifndef _SYSTAB_H_
#define _SYSTAB_H_

#include "linklist.h"
#include "distproc.h"
#include "syserr.h"

#define ulong unsigned long

template <class T>
class SystemTable
{
public:
     int Insert (T &item);
     int Update (T &item);
     int Delete (T &item);
     int Find (T &item);
     int FindFirst (T &item);
     int FindNext (T &item);

protected:
     LinkedList<T> list;
};

template <class T>
int SystemTable<T>::Insert (T &item)
{
     list.AppendEntry(item);

     return GOOD;
}

template <class T>
int SystemTable<T>::Update (T &item)
{
     if (Delete(item) == FAILED)   // remove old item from list
          syswarn("Update(), non-existant item");

     return Insert(item);          // add again
}

template <class T>
int SystemTable<T>::Delete (T &item)
{
     T *cur;

     //
     // go through list looking for id
     //
     cur = list.FirstEntry();
     while (cur != NULL)
          if (*cur == item)
               {
               list.RemoveEntry();
               return GOOD;
               }
          else
               cur = list.NextEntry();

     return FAILED;
}

template <class T>
int SystemTable<T>::Find (T &item)
{
     T *cur;

     cur = list.FirstEntry();
     while (cur != NULL)
          if (*cur == item)
               {
               item = *cur;
               return GOOD;
               }
          else
               cur = list.NextEntry();

     return FAILED;
}

template <class T>
int SystemTable<T>::FindFirst(T &item)
{
     T *cur;

     cur = list.FirstEntry();
     if (cur == NULL)
          return FAILED;

     item = *cur;
     return GOOD;
}

/*
;  On entry, item is the current item in the table.
;  On exit, item is the next item in the table.
*/
template <class T>
int SystemTable<T>::FindNext (T &item)
{
     T *cur;

     if (Find(item) == FAILED) // set position at current entry
          return FAILED;
     if ((cur=list.NextEntry()) == NULL) // go to next entry
          return FAILED;

     item = *cur;
     return GOOD;
}

#endif   /* #ifndef _SYSTAB_H_ */
```

```
/*
     tables.h
     Brian Cox.
     September 1993.

     include file for tables.cpp
*/

#ifndef _TABLES_H_
#define _TABLES_H_

#define MSDOS

#ifdef __cplusplus
extern "C" {
#endif

#include "gdbm.h"


#include <stdio.h>

extern gdbm_error gdbm_errno;
extern char * gdbm_version;

/*
 The following typedef is only used for DOS.3
 typedef gdbm_file_info* GDBMFILE;
*/
#define GDBMFILE GDBM_FILE


/* function declarations */
int   tblCreateTable (GDBMFILE *dbf, char *name, void
(*fatal_func)(), int block_size);
int   tblOpenTable (GDBMFILE *dbf, char *name, void
(*fatal_func)(), int block_size);
int   tblCloseTable (GDBMFILE dbf);
int   tblInsertEntry (GDBMFILE dbf, datum key, datum data);
int   tblUpdateEntry (GDBMFILE dbf, datum key, datum data);
int   tblDeleteEntry (GDBMFILE dbf, datum key);
datum tblFindEntry (GDBMFILE dbf, datum key);
datum tblFindFirst (GDBMFILE dbf);
datum tblFindNext (GDBMFILE dbf, datum key);

/*
** the following functions are used to serialise access
** to a table.  Not implemented yet - use semaphores.
*/
/*
int tblLockTable(GDBMFILE dbf);
int tblUnlockTable(GDBMFILE dbf);
*/

#ifdef __cplusplus
}
#endif

#endif /* #ifndef _TABLES_H_ */
```

```
/*
     tcp.h
     Brian Cox.
     August 1993.
     Revised:  Sept. 1993, cleaned up functions.
     Client/Server Programming with OS/2, chapter 16, page 361.
*/

#ifndef _TCP_H_
#define _TCP_H_

/*
** TCP/IP include files.
*/
#include <unistd.h>       /* close() */
#include <sys/types.h>
#include <sys/param.h>    /* MAXHOSTNAMELEN */
#include <sys/socket.h>
#include <sys/select.h>   /* select() */
#include <sys/time.h>     /* timeval for tout */
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>        /* gethostname() */

#include "syserr.h"


/*
** Forward declarations of communication functions.
*/

extern int InitPipe ();
extern int MakePipe (int *, short, int);
extern int WaitConnectPipe (int, int *, struct sockaddr_in *, int
* );
extern int DisconnectPipe(int);
extern int OpenPipe(int *, short, char *);
extern int WritePipe(int, void *, unsigned short);
extern int ReadPipe(int, void **, unsigned long *, int tout);
extern int TransactPipe(int, void *, unsigned short, void **,
unsigned long *, int tout);
extern int ClosePipe(int);

#endif  /* #ifndef _TCP_H_ */
```

```
/*
     waitinfo.h
     Wait System Information class.
     Brian Cox
*/

#ifndef _WAITINFO_H_
#define _WAITINFO_H_

#include "distproc.h"
#include "buffer.h"
#include "saveobj.h"

#define ulong unsigned long

/*---------------------------------------*/
/* record structure of wait system table. */
/*---------------------------------------*/
class WaitInfo
{
public:
     WaitInfo ();
     WaitInfo (const WaitInfo &source);
     WaitInfo& operator = (const WaitInfo &source);
     int operator == (const WaitInfo &source);

     int    save (Buffer &b);
     Buffer load (const Buffer& b);

     ulong getPID ();
     int   getSock ();

     void setPID (ulong id);
     void setSock (int sock);

protected:
     ulong  pid;      // id of RemProc to wait for
     int    socket;  // socket on which request was received.
};

#endif  /* #ifndef _WAITINFO_H_ */
```

```
/*
     waitrm.h
     Brian Cox.
     System tables for both WaitProcess and WaitAnyProcess
requests.
*/

#ifndef _WAITRM_H_
#define _WAITRM_B_

#include "distproc.h"
#include "waitinfo.h"
#include "systab.h"

class WaitResMgr : public SystemTable<WaitInfo>
{
public:
     /*
     ;  WaitProcess requests.
     */
     int Add (ulong pid, int sock);
     int Get (ulong pid, int &sock);
};


/*
** WaitAnyProcess requests..........
*/
//int AddWaitAnyRemProc (char pname[MAX_LEN_PNAME], int sock);
//int GetWaitAnyRemProc (char pname[MAX_LEN_PNAME], int &sock);

#endif /* #ifndef _WAITRM_H_ */
```

8.8 Example

```
/*
;   DPLenstra: Main DistProc Process
;
;   Program to factor big numbers using Lenstras elliptic curve
method.
;   Works when for some prime divisor p of n, p+1+d has only
;   small factors, where d depends on the particular curve used.
*/

#include <iostream.h>
#include <fstream.h>
#include <iomanip.h>
#include <time.h>
#include "number.h"
#include "big.h"
#include "dpclient.h"

//
// Qn = (10^n)+1
// divideby is used to get rid of small factors
//

//#define n          "11"
//#define divideby  (1)

//#define n            "19"
//#define divideby     (1)   // factors are: 11 .
909090909090909091

//#define n          "24"
//#define divideby  (17)

//#define n           "37"
//#define divideby   (11*7253)

//#define n          "41"
//#define divideby  (11)

#define n          "59"
#define divideby  (11*1889)


//#define Limit1  2000            // must be int, and > MULT/2
//#define Limit2  100000L         // may be long

#define MULT     210              // must be int, product of small
primes 2,3...
#define NCURVES  5000             // number of curves to try
#define LOGb2E   1.442695041

miracl precision(50,MAXBASE);     // number of ints per ZZn

ulong  Limit1, Limit2;            // limits for the curves
static long p;
ulong  rpid[NCURVES];             // list of RemProc ids
Buffer resbuf[NCURVES];           // results for each RemProc are
stored here
int NumSlaves, MaxRemProcs;       // # of DPSlaves and max # of
RemProcs
DPClient *dp;                     // DPClient requester object

class Timings
{
public:
     time_t stm, etm, setupruntm, waittm;
};

Timings tm[NCURVES];
ofstream logfile;   // create & open log file

//---------------------------------------------------------
// Pack the parameters and run the rplenstra process
//---------------------------------------------------------
unsigned long RunRPLenstra (int curve, Big &num, Buffer *result)
{
     unsigned long rpid;
     Buffer inbuf;

     SAVE(inbuf, curve);
     cotstr(num.fn, (char *)MR_IOBUFF);
     SAVE(inbuf, (char *)MR_IOBUFF, (size_t)MR_IOBSIZ);

     rpid = dp->run("rplenstra", inbuf, result);

     if (rpid == INVALID_DPID)
          {
          logfile << "\nCouldn't run rplenstra process\n\n" <<
flush;
          cout << "\nCouldn't run rplenstra process\n\n" << flush;
          }

     return rpid;
}

//---------------------------------------------------------
// Wait for one of the rplenstra processes to finish.
//---------------------------------------------------------
int WaitRPLenstra(int p, int &retcode, Big &t)
{
     int rc;

     if ((rc = dp->wait(rpid[p])) == GOOD)
          {
          resbuf[p] = LOAD(resbuf[p], retcode);
          resbuf[p] = LOAD(resbuf[p], (char *)MR_IOBUFF,
(size_t)MR_IOBSIZ);
          cinstr(t.fn, (char *)MR_IOBUFF);

          logfile << "\nReceived result from RPLenstra #" << p <<
"\n";
          cout << "\nReceived result from RPLenstra #" << p <<
"\n";
          }
     else
          {
          logfile << "\n+++ Error waiting for RPLenstra #" << p <<
" +++\n";
          cout << "\n+++ Error waiting for RPLenstra #" << p << "
+++\n";
          }

     return rc;
}

void InitDPClient (int argc, char *argv[])
{
     //
     // Initialise the DistProc Client requester
     //
     if (argc != 3)
          {
          cerr << "usage: progname DPServer NumSlaves.\n";
          exit(-1);
          }

     dp = new DPClient(argv[1]);
     if (dp->Registered() == FAILED)
          exit(-1);

     NumSlaves = atoi(argv[2]);
     MaxRemProcs = NumSlaves;     // One RemProc per DPSlave!!!
}

//--------
// main()
//--------
int main(int argc, char *argv[])
{
     int r, w, curve, rc, power;
     Big num, t;
     time_t t1, t2, st, et;
     char logname[20];

     strcpy(logname, "dpl_");
     strcat(logname, n);
     strcat(logname, ".log");
     logfile.open(logname, ios::app);

     time(&t1);              // Get start time.
     logfile << "Start Time:" << ctime(&t1) << flush;

     InitDPClient(argc, argv);

     power = atoi(n);
     num = 10;
     num = (pow(num, power) + 1) / divideby;
     logfile << "\n\nTrying to factor ((10^" << power << ")+1)\n"
<< flush;

     //
     // calculate limits, check if num is prime
     // and do all arithmetic mod num
     //
     Limit1 = (unsigned long)(pow(bits(num)/LOGb2E, 2.65)/10.0);
     Limit2 = 40 * Limit1;
     gprime(Limit1);
     if (prime(num))
          {
          logfile << "This number is prime!\n" << flush;
          logfile.close();
          cout << "This number is prime!\n";
          exit(0);
          }
     modulo(num);                    // do all arithmetic mod n

     //
     // Now distribute the curves using the distproc library
     //
     for (r=0, curve=3; r<NCURVES && r<MaxRemProcs; r++, curve++)
          {
          time(&(tm[r].stm));

          rpid[r] = RunRPLenstra(curve, num, &resbuf[r]);

          time(&et);
          tm[r].setupruntm = difftime(et, tm[r].stm);
          }

     cout << "\nWaiting for the processes to finish...\n" <<
flush;

     //
     // Now wait for each remote process to finish.
     // Check each result as it comes in.
     // The remproc returns a number 't' which is a factor.
     // If 't' is 0 [zero] then no factor was found
     //
     for (w=0; w<NCURVES; w++)
          {
          time(&st);

          rc = WaitRPLenstra(w, rc, t);

          time(&(tm[w].etm));
          tm[w].waittm = difftime(tm[w].etm, st);

          if (rc != FAILED)
               if (t == 0) // || rc != 0)
                    {
                    logfile << "Curve " << w+3 << " failed\n" <<
flush;
                    cout << "Curve " << w+3 << " failed\n" <<
flush;
                    }
               else
                    break;   // -------- Factor found, exit loop --
-----
```

```
        //
        // Try and start another RemProc running.
        //
        if (r<NCURVES)
            {
            time(&(tm[r].stm));
            rpid[r] = RunRPLenstra(curve, num, &resbuf[r]);

            time(&et);
            tm[r].setupruntm = difftime(et, tm[r].stm);

            curve++;
            r++;
            }
        }

    //
    // close the DP Client object
    //
    delete dp;

    //
    // Print out the factors
    //
    if (t != 0)
        {
        logfile << "\n\nFactors of " << num << " = (10^" <<
power << ")+1"
              << "\n\nfound with curve #" << w+3 << endl
              << "Limit1 = " << Limit1 << " and Limit2 = " <<
Limit2 << endl;

        cout << "\n\nFactors of " << num << " = (10^" << power
<< ")+1"
            << "\n\nfound with curve #" << w+3 << endl
            << "Limit1 = " << Limit1 << " and Limit2 = " <<
Limit2 << endl;

        if ( prime(t) )
            {
            logfile << "\nprime factors " << t << flush;
            cout << "\nprime factors " << t;
            }
        else
            {
            logfile << "\ncomposite factor " << t << flush;
            cout << "\ncomposite factor " << t;
            }

        num /= t;

        if ( prime(num) )
            {
            logfile << "\nprime factor (num = num/t) " << num;
            cout << "\nprime factor (num = num/t) " << num;
            }
        else
            {
            logfile << "\ncomposite factor (num = num/t) " <<
num << flush;
            cout << "\ncomposite factor (num = num/t) " << num;
            }
        }

    //
    // Print out the timings
    //
    time(&t2);
    logfile << "\n\nStart Time: " << ctime(&t1)
           << "Finish Time: " << ctime(&t2)
           << "Running Time: " << difftime(t2, t1)/60 << "
minutes.\n"
           << "There were " << MaxRemProcs << " DPSlaves
used.\n"
           << flush;

    cout << "\n\nStart Time: " << ctime(&t1)
         << "Finish Time: " << ctime(&t2)
         << "Running Time: " << difftime(t2, t1)/60 << "
minutes.\n"
         << "There were " << MaxRemProcs << " DPSlaves used.\n"
         << flush;

    logfile << "curve  (Start Time)  (End Time)  (Run Setup Time)
(Wait Time)\n\n";
    for(int i = 0; i <= w; i++)
        logfile << i+3 << "\t" << tm[i].stm << "\t" << tm[i].etm
<< "\t"
              << tm[i].setupruntm << "\t" << tm[i].waittm <<
endl;
}
```

```
/*
;   Lenstra Remote DistProc Process
;
;   Program to factor big numbers using Lenstras elliptic curve
method.
;   Works when for some prime divisor p of n, p+1+d has only
;   small factors, where d depends on the particular curve used.
;   See "Speeding the Pollard and Elliptic Curve Methods"
;   by Peter Montgomery, Math. Comp. Vol. 48 Jan. 1987 pp243-264
*/

#include <iostream.h>
#include <iomanip.h>
#include "number.h"
#include "big.h"
#include "rpargio.h"           // Remote Process ARGument I/O
header file


//#define LIMIT1 2000          // must be int, and > MULT/2
//#define LIMIT2 10000000L     // may be long
#define MULT   210             // must be int, product of small
primes 2.3...
#define NCURVES 20             // number of curves to try
#define LOGb2E   1.442695041
miracl precision(50,MAXBASE); // number of ints per ZZn


long LIMIT1, LIMIT2;
static long p;
static int iv;
static ZZn ak,q,x,z,x1,z1,x2,z2,xt,zt,fvw,fu[1+MULT/2];
static bool cp[1+MULT/2];

/*
  double a point on the curve P(x,z)=2.P(x1,z1)
*/
void duplication(ZZn sum,ZZn diff,ZZn& x,ZZn& z)
{
    ZZn t;
    t=sum*sum;
    z=diff*diff;
    x=z*t;         /* x = sum^2.diff^2 */
    t-=z;          /* t = sum^2-diff^2 */
    z+=ak*t;       /* z = ak*t +diff^2 */
    z*=t;
}

/*
  add two points on the curve P(x,z)=P(x1,z1)+P(x2,z2)
  given their difference P(xd,zd)
*/
void addition(ZZn xd,ZZn zd,ZZn sm1,ZZn df1,ZZn sm2,ZZn df2,ZZn&
x,ZZn& z)
{    ZZn t;
    x=df2*sm1;
    z=df1*sm2;
    t=z+x;
    z-=x;
    x=t*t;
    x*=zd;   /* x = zd.[df1.sm2+sm1.df2]^2 */
    z*=z;
    z*=xd;   /* z = xd.[df1.sm2-sm1.df2]^2 */
}


/*
  calculate point r.P(x,z) on curve
*/
void ellipse(ZZn x,ZZn z,int r,ZZn& x1,ZZn& z1,ZZn& x2,ZZn& z2)
{
    int k,rr;
    k=1;
    rr=r;
    x1=x;
    z1=z;
    duplication(x1+z1,x1-z1,x2,z2);  /* generate 2.P */
    while ((rr/=2)>1) k*=2;
    while (k>0)
    { /* use binary method */
        if ((r&k)==0)
        { /* double P(x1,z1) mP to 2mP */
            addition(x,z,x1+z1,x1-z1,x2+z2,x2-z2,x2,z2);
            duplication(x1+z1,x1-z1,x1,z1);
        }
        else
        { /* double P(x2,z2) (m+1)P to (2m+2)P */
            addition(x,z,x1+z1,x1-z1,x2+z2,x2-z2,x1,z1);
            duplication(x2+z2,x2-z2,x2,z2);
        }
        k/=2;
    }
}


/*
  now change gear
*/
void next_phase()
{
    ZZn s1,d1,s2,d2;
    xt=x;
    zt=z;
    s2=x+z;
    d2=x-z;                         /* P = (s2,d2) */
    duplication(s2,d2,x,z);
    s1=x+z;
    d1=x-z;                         /* 2.P = (s1,d1) */
    fu[1]=x1/z1;
    addition(x1,z1,s1,d1,s2,d2,x2,z2);  /* 3.P = (x2,z2) */
    for (int m=5;m<=MULT/2;m+=2)
    ( /* calculate m.P = (x,z) and store fu[m] = x/z */
        addition(x1,z1,x2+z2,x2-z2,s1,d1,x,z);
        x1=x2;
        z1=z2;
        x2=x;
        z2=z;
        if (!cp[m]) continue;
        fu[m]=x2/z2;
    }
    ellipse(xt,zt,MULT,x,z,x2,z2);
    xt=x+z;
    zt=x-z;                             /* MULT.P = (xt,zt) */
    iv=p/MULT;
    if (p%MULT>MULT/2) iv++,p=2*(long)iv*MULT-p;
    ellipse(x,z,iv,x1,z1,x2,z2);        /* (x1,z1) = iv.MULT.P */
    fvw=x1/z1;
    q=fvw-fu[p%MULT];
}


/*
  increment giant step
*/
int giant_step()
{
    iv++;
    p=(long)iv*MULT+1;
    fvw=x2/z2;
    addition(x1,z1,x2+z2,x2-z2,xt,zt,x,z);
    x1=x2;
    z1=z2;
    x2=x;
    z2=z;
    return 1;
}


int lenstra(Big &n, int k, Big& t)
{
    int phase, m, nc, pos, btch;
    long i,pa;
    ZZn tt;
     Big x;

     //
     // calculate limits...
     //
     LIMIT1 = ((long)pow(bits(n)/LOGb2E, 2.65)) / 10.0;
     LIMIT2 = 40 * LIMIT1;
     if (LIMIT1 <= MULT/2)
          LIMIT1 = (long)(MULT/2) +1;
     cout << "\nLimit1 = " << LIMIT1
          << "\nLimit2 = " << LIMIT2 << endl << flush;

    gprime(LIMIT1);
    for (m=1;m<=MULT/2;m+=2)
        if (igcd(MULT,m)==1)
             cp[m]=TRUE;
        else
             cp[m]=FALSE;

     modulo(n);                  /* do all arithmetic mod n */

     /* try a new curve */
     x=2;          /* generating an elliptic curve  */
     z=1;
     nc++;
     tt=4-k*k;
     tt/=(2*(k*k-1));
     tt+=(1/tt);
     ak=(tt+2)/(ZZn)4;
     phase=1;
     p=0;
     i=0;
     btch=50;

     cout << "phase 1 - trying all primes less than " << LIMIT1;
     cout << "\nprime= " << setw(8) << p;


    /* main loop */
    forever
        {
         if (phase==1)
            {
             p=PRIMES[i];
             if (PRIMES[i+1]==0)
                  { /* now change gear */
                  phase=2;
                  cout << "\nphase 2 - trying last prime less
than ";
                  cout << LIMIT2 << "\nprime= " << setw(8) << p;
                  next_phase();
                  btch*=10;
                  i++;
                  continue;
                  }
             pa=p;
             while ((LIMIT1/p) > pa) pa*=p;
             ellipse(x,z,(int)pa,x1,z1,x2,z2);
             x=x1;
             q=z=z1;
             }
         else
             { /* looking for last large prime factor of (p+1+d)
*/
             p+=2;
             pos=p%MULT;
             if (pos>MULT/2) pos=giant_step();
             if (!cp[pos]) continue;
             q*=(fvw-fu[pos]);         /* batch gcds */
             }
         if (i++%btch==0)
             {
             /* try for a solution */
             cout << "\b\b\b\b\b\b\b\b" << setw(8) << p <<
flush;
             t=gcd(q,n);
             if (t==1)
                  if (p>LIMIT2) break;
                  else continue;
             if (t==n)
                  {
                  cout << "\ndegenerate case";
                  break;
                  }
```

```
                if (prime(t)) cout << "\nprime factor     " << t;
                else          cout << "\ncomposite factor " << t;
                n/=t;
                if (prime(n)) cout << "\nprime factor     " << n;
                else          cout << "\ncomposite factor " << n;
                return GOOD;
                }
            }

     t = 0;              // to signify that we didn't find a
factor!
    return FAILED;
}


//--------------------------------------------------------
// factoring program using Lenstras Elliptic Curve method
//--------------------------------------------------------
int main()
{
     Buffer inbuf, outbuf;
     Big num, factor;
     int curve, rc;

     //
     // get input parameters from the main distproc process
     //
     GetInputParameters(inbuf);
     inbuf = LOAD(inbuf, curve);   // load curve
     inbuf = LOAD(inbuf, (char *)MR_IOBUFF, (size_t)MR_IOBSIZ);
     cinstr(num.fn, (char *)MR_IOBUFF);

     //
     // call our worker function to factor 'n' using curve 'k'
     // and put the result into 'factor'.
     //
     rc = lenstra(num, curve, factor);

     //
     // send result back to the main distproc process
     //
     SAVE(outbuf, rc);
     cotstr(factor.fn, (char *)MR_IOBUFF);
     SAVE(outbuf, (char *)MR_IOBUFF, (size_t)MR_IOBSIZ);
     ReturnResults(outbuf);

     cout << "\n\nReturned the following results:\n"
          << "\n\trc = " << rc
          << "\n\tfactor = " << factor << "\n\n";

     return 0;
}
```