THE SHIP MODEL – A NEW COMPUTATIONAL MODEL FOR DISTRIBUTED SYSTEMS

By

George William Phillips

B. Sc. (Computer Science) University of British Columbia, 1987

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in

THE FACULTY OF GRADUATE STUDIES

COMPUTER SCIENCE

We accept this thesis as conforming

to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA

1992

© George William Phillips, 1992

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Computer Science
The University of British Columbia
2075 Wesbrook Place
Vancouver, Canada
V6T 1Z1

Date: October 14, 1992

Abstract

One of the fundamental goals of a distributed operating is to make a collection of computers connected by a network appear as a unified whole to the users of the system. The system relies heavily on the network to help maintain this illusion. If the network is not fast enough system performance will be noticeably affected and the system will fail to achieve this primary goal. Since the network is used to allow two entities on different machines to communicate with each other, it is possible to reduce the use of the network by allowing entities to migrate between machines. Two entities on the same machine will not require the network for their communication. A distributed system which supports such a mobility mechanism has two primary benefits. First, it can increase the performance and usability of the system since decreased network communication can generally increase performance. The performance gain is most notable over low speed networks where decreased use of the network is vital for the system to perform adequately. Second, it also makes the design of system interfaces simpler by removing features from the design that are necessary only for good remote interaction. This thesis investigates a mechanism that allows distributed programs to reduce their network usage by moving code segments between computers. The general idea of moving code is developed into an abstract model of distributed computation call the Ship Model. The Ship Model has a basic entity called a *ship* which contains code and data. It uses mobility both as a way of moving code and data between machines and as an inter-ship communication mechanism. In order to test the viability of the model a prototype implementation is constructed. The prototype is written in C and runs under UNIX with the two parts of the system connected either by a high speed ethernet network or a slow speed dialup line. A few example applications are implemented and tested under the prototype. Various measurements show that for many applications the Ship Model can provide increased performance in a high speed network setting as well as over a slow speed line. Various issues regarding the implementation of the Ship Model are also discussed.

Table of Contents

List of Tables

List of Figures

Acknowledgement

I would like to thank my supervisor, Sam Chanson without whose support this thesis could not have been finished. I also thank Norm Hutchinson for his work as the second reader.

I would also like to thank my wife, Tarmi, who has had the patience to deal with my absences and prodded me to keep going.

Finally a thank-you to all those who have helped along the way, especially those who have resisted enquiring on the state of my thesis.

Chapter 1

Introduction

Computer systems are constantly becoming more interconnected. Phone lines and modems connect home computers. Office computers are joined together with high speed local area networks. Long haul networks connect universities, businesses and governments all over the world. The trend is to make the standalone computer the exception rather than the rule. Loosely speaking, these collections of computers connected by some communication link are referred to as distributed systems. The basic goal of a distributed system is to facilitate information sharing between the computers in the system. This goal relies on physical hardware which lets the computers communicate at some physical level. The hardware may be a slow dialup line which provides bandwidth on the order of hundreds of characters per second or optically based cabling providing billions of characters per second. In either case the distributed system will build some mechanism on the communications hardware. The system may provide only weak interconnection that requires the user to carefully direct which other computer(s) to connect his computer to and what specific information is to be transferred. Computers linked with dialup lines usually operate in this fashion as do most computers connected over long haul networks such as the internet. The system might also provide strong interconnection between its computers. Here the user can interact with many systems without even being aware of the physical separation of the machines. Indeed, the distributed system appears as a single computer. Strongly interconnected distributed systems usually operate on a small scale, typically encompassing a floor, a building or at most a few buildings, with high speed communication links delivering on the order of millions of bytes per second.

A distributed operating system [11] is an operating system for a distributed system that makes the system appear as a unified whole. System resources such as files or printers are accessible on every machine on the system regardless of their actual location. A uniform view of the system is desirable; the user will only be hampered by restrictions on where certain operations can be performed. A distributed operating system makes location unimportant. The location of the user and the location of the desired system resources do not have any effect on the user's ability to access those resources. The system keeps track of resource location and makes sure that requests are directed to the appropriate machines. Most distributed systems preserve the user's uniform view at the programmer's level as well. This is a practical necessity since the alternative would be to have every system application work to preserve the illusion of location independence. The programmer uses the same routines to access both local and remote resources such as files or other programs. The lowest levels of the system determine whether local or remote accesses are needed. Support within the operating system removes the burden from the applications programmer who can easily create location transparent programs.

There are some essential difficulties faced by distributed operating system designers. Above all the system must be reliable. For all intents and purposes a single computer is either running correctly or it is down. A distributed system can exhibit partial failures, that is, particular computers or networks in the system can fail while others remain up. A good distributed operating system must be able to make progress in the face of partial failures. Reliability is also affected by the communication links between computers in the system. The hardware will not provide absolutely reliable data transfer between computer systems so the operating system must provide error recovery protocols to ensure reliable data transfer. Performance is also a concern. The illusion of a single system will not be very convincing or useful if access to remote resources is noticeably much slower. Users will work to thwart remote accesses thereby negating the supposed benefit of the distributed operating system. Fortunately, local area networks such as ethernet mitigate the reliability and performance problems. They provide very low error rates which essentially reduce reliability problems to those of individual computer failures. They also provide high data rates which are sufficient to make remote accesses and transfer of data fast enough that most applications run locally and remotely with little perceived difference.

However, fast, reliable networks do little to help problems associated with heterogeneity of computer architectures. Most modern operating systems, distributed or not, present a uniform programmer interface to the system even across different hardware platforms [1, 7]. The programmer only needs to re-compile the source code to get a binary which can run on a different system. But once compiled the program can only be run on some subset of the computer systems in a distributed system. Tying programs to architectures in this way removes some of the transparency of the distributed operating system since the user may not always be able to run a particular program. Heterogeneity remains a largely unsolved problem in distributed operating system research. Research into both interpreters and compilers is useful in approaching the problem, but performance degradation is unavoidable.

The location of system resources is an important factor in a distributed system. Location affects reliability. Access to resources can be lost if they do not reside locally. Location also affects performance. It will generally be slower to access a remote rather than a local resource. Even on a high speed network the performance drop can be noticable. On a low speed network the performance drop can be devastating. While it is in principle enough for the distributed operating system to provide remote access to a resource, in practice low speed networks, large data transfers and frequent interactions demand extra support. Some way of speeding up the communication is needed. The direct solution is to get a faster network, but such upgrades are difficult even when they are possible. If the network can not be made faster, then the resource and the program must be brought closer together to speed up their communication. One way is to replicate the resource at multiple locations so that the program will always have some copy nearby. Replication can work well, but it can not always be used since some resources like user input and output devices are too tied to a physical presence to be replicated. Another possibility would be to move the resource so that it is close to the program. This technique is similar to replication and suffers the same drawbacks. The only other approach is to move the program closer to the resource. While heterogeneous machines are a large obstacle to this approach, even migrating code between homogeneous machines (machines of the same architecture) is difficult [15, 13, 16]. Any scheme which allows resource or code migration to improve performance must face a general problem of cost balancing. Will performance actually increase if a program or resource is moved? It may not be possible for the system itself to answer this question, but even programs that are directly involved will have difficulty deciding if something should move or not. While using resource or program mobility to increase performance is difficult, it can provide performance increases over high speed networks and actually make the operation of a distributed operating system over low speed lines possible.

This thesis investigates the problem of resource location in distributed operating systems. A model of distributed computation, the Ship Model, is presented as a solution to the problem of resource location. This model is used in the construction of a prototype distributed system which can operate over heterogenous machine architectures. Various measurements of the prototype show the viability of the model both in high speed and, most importantly, low speed network settings.

Chapter 2

Motivations

Communication overhead can be greatly increased if the location of resources and programs are fixed. A good example of this is file copying. There are three important locations when copying a file: the sites of the source file, the destination file and the copying program. If all the locations are the same, the work done is entirely local. But in this age of diskless workstations and file servers, it is often the case that the program must run where none of the files are placed. Worse yet, the files are often at the same location, on the file server for example. In this case the file is sent from the server to the copy program and right back to the server. The natural solution to this problem is to break the copy program into two parts. One part, the reader, reads the source file and sends the data to the second part, the writer, which writes the data to the destination. The operation of the copy program is to send the reader to the site of the source file and the writer to the site of the destination file. This arrangement directly provides the optimal communication path and gives the ideal case when presented with the common case of a diskless workstation and a single file server. While any extra copying done is disastrous only in a low speed setting, even high speed networks pay a price.

Designing a network resource is often a difficult struggle between making a simple interface and providing good performance. Typical virtual terminal interfaces suffer this fate. A call to read a character at a time would be sufficient for input, but must be avoided because of the network overhead of transmitting a character at a time. Instead, a line mode is provided and a block of characters is sent when a carriage return is entered.

But line mode is not good enough for a full screen editor which requires editing keys to be transmitted immediately. More calls are added to the interface to describe which keys must be sent immediately and which can be buffered. A tempting solution would be to embed a programming language into the terminal interface. Programs wishing line mode input would send a small program that reads characters until a newline and passes the complete line back at once. Programs such as editors can send more complex programs that interact with the keyboard and display and greatly reduce the number of characters sent back to the host. Putting all this functionality into a virtual terminal protocol is clearly incorrect. Using some separate mechanism which allows programs to change the granularity of a network resource by transferring code to enhance the interface is a better approach. This technique allows the interface to a resource to be as simple as possible without degrading the performance of programs accessing the resource remotely. The terminal only needs interfaces to read and write characters. Code moved near the terminal can enhance the interface to provide line modes and other enhanced interfaces necessary for good interactive response. Having programs move code to the terminal gives us another advantage. A "novice" mistake that infuriates system administrators is the "rlogin" chain. A user logs into a machine and then remotely logs into another, and then remotely logs into another from there and so on. A number of machines and the network experience extra load from shuffling this user's characters among themselves. If each shell were to talk directly to the user's original terminal (by moving code there) instead of the interposed pseudo-terminals, the communication graph would collapse into a one level tree.

A primary motivation for minimizing network usage is to permit a distributed operating system to utilize low speed communication lines. A possible criticism of any system that supports low speed communication is that low speed lines will go away. There are a number of responses to this claim. First, low speed lines will always be with us. The advent of ISDN and broadband ISDN will eliminate them from some areas, but they will still be around in many places for a long time. Even then the connections to a residence will still be slower than the networks at places of work. Second, a model which works well at low speed has benefits for high speed networks as well. Conservation of bandwidth is an easier way to increase the capability of a network than by installing more wires. Third, similar arguments were proposed against virtual memory in the sixties and seventies. It was argued that main memory would become so large that virtual memory would be unnecessary; everything would fit in main memory. Virtual memory is still thriving since software requirements have easily kept pace with increases in memory size. Similarly, software's usage of network bandwidth will continue to increase. Reducing that usage will always be important. Finally, modern physics dictates that there are ultimate speed limits. Propagation delays can only be reduced to a certain point. Schemes which use code mobility capitalize on locality of reference will ultimately hold advantages over those that do not.

A review of related work is given in Chapter 6.

Chapter 3

The Ship Model

The basic element of the Ship Model is the *ship*. A ship contains code and data and represents a location. The code within a ship may have zero or more threads of control. Ships are mobile and can move between locations. When a ship is at a particular location, it is inside the ship which represents that location. While a ship is inside another ship, it has access to the code and data of the containing ship. A ship can communicate with another by moving into it and writing information into the data space of the ship. Protection and security problems associated with this mode of operation will be addressed in section 3.2.

Using ships as the container of a location allows fine control over when to differentiate between locations. While it is up to the implementors to determine the boundaries of locations, a good rule of thumb would be to differentiate two locations when communication between them is significantly costly. Different hosts on a LAN, two nodes in a multiprocessor, or even virtual address spaces on the same machine may be different locations.

While some ships correspond to the traditional chunks of code and data we call programs, others correspond to entities with roots in the physical world. Hosts, disks, screens, keyboards and other pieces of hardware will be represented by ships. While it is possible to move such ships, there are a number of substantial constraints on their movement. Storage devices would have to be moved onto some other storage device. Moving a keyboard or screen will also require informing the user to move as well. With these heavily constrained ships, it may be just as well to consider them immobile. There are a number of other reasons that we may decide to make a ship immobile, but those depend on the particular usage of the Ship Model.

3.1 Primitives

A list of primitives used by ships to implement their functions is listed below. The primitives are presented as function calls, but the actual interface will obviously depend on the implementation.

move(*sid*)

This primitive moves the calling ship into the specified ship (indicated by the ship ID *sid*). If *sid* is the special ID "oblivion", the calling ship is destroyed. Any ships contained in the calling ship are moved with it.

fork() → *thread_ID*

This call creates another thread of control in the calling ship. It returns the created thread's ID to the calling thread and an invalid thread ID to the new thread.

kill(*thread_ID*)

Terminate the given thread of execution.

self(*which*) → *thread_or_ship_ID*

Returns the ID of either the calling thread if *which* is 0, or the ID of the calling ship if *which* is 1.

commission(*code_ptr*, *data_ptr*) → *ship_ID*

This call creates a new ship within the calling ship. The code and data for the new ship are indicated by the code and data pointers. The calling ship is returned the

ID of the created ship. The created ship has a single thread of control which begins execution at the start of the code. A ship with no threads of control can have the thread execute a kill(self(0)) call immediately.

lock(*lock_ship*, *entry_ship*)

A synchronization primitive which prevents every ship except *entry_ship* from moving to *lock_ship*. Ships attempting to move into or lock a locked ship will be blocked.

unlock(*lock_ship*)

The call removes the lock on *lock_ship* and lets any ship move to it. Any ships which were blocked because *lock_ship* was locked are resumed.

block(*lock_ship*)

The calling ship voluntarily blocks on *lock_ship*. It will be unblocked when the *lock_ship* is unlocked by some ship executing the unlock primitive.

These primitives provide a basis for a working system. The move primitive is the most important; it is the sole way in which ships can interact. In order for two ships to exchange data, one of them must move into the other. Thus the move primitive unifies two distinct operations, inter-ship communication and ship mobility. If the destination ship in a move is at a different location, the calling ship will first be moved to be close to the destination and then into the destination. When the move is complete, the ships may then exchange information since each will have access to the other's code and data spaces. Exactly how the ships interact is not specified since that will depend on the implementation.

Primitives for creating new threads and new ships are also necessary. Any non-trivial program written under the Ship Model will consist of many cooperating ships.

Such a program will need to create new ships and threads as necessary to complete its task. Ship creation will be used frequently in order to exploit the Ship Model's ability to minimize remote communication. If a ship needs to perform some operation it can move to another ship and do so. However, the ship moved will likely contain extra code and data not pertinent to the remote operation. It can instead create a new ship which contains only the code necessary to the task thereby reducing the communication needed.

Any system with multiple processes needs some form of synchronization. The Ship Model provides these with its lock, unlock and block primitives. Lock and unlock allow cooperating ships to access a ship one at a time. The locked ship can encapsulate some critical section or can serve as a synchronization point for some other operation. The block primitive enhances a ship's ability to synchronize. If a collection of ships merely needs to guarantee that they move into some ship in a serial fashion, they can use lock and unlock. If each movement must occur in a particular order it is necessary to use block. Each ship may enter the destination ship and check to see if it is its turn to run. If so, it can continue; otherwise it will use block to wait until some other ship has updated the conditions and signalled it has done so with unlock.

Some systems which allow for mobility provide an operation to make the system guarantee that two objects will always be together at some location. Two objects will typically desire guaranteed proximity if they plan on performing expensive or frequent interactions whose efficiency would be severely hampered should the objects move apart. Such a primitive is unnecessary in the Ship Model. The very nature of the move primitive is to put two ships in close proximity. Since there is only a single mechanism for ship interaction, it is not possible for two interacting ships to move away from each other.

It is entirely possible that a distributed program may wish to know the communication cost between locations so that it may make dynamic decisions about its distribution amongst some locations. A primitive for measuring the "distance" between two locations could have been added. However, a Ship program could empirically measure the cost itself by sending a ship on a round trip between locations. This is the preferred solution since it saves adding a primitive and the empirical measurement technique will give more than adequate results.

3.2 Implementation Issues

There are a number of questions to be considered before a system or even partial implementation can be built. A representation for the ship code will be needed along with protection mechanisms for ships. It is clear that moving a ship within a ship provides inter-ship communication, but an exact method to do this must be described.

The basis of a system is the ships that represent hardware. A file ship will abstractly represent the disk. Screen, keyboard and mice ships can represent some basic I/O devices. Host ships would represent specific machines and so on. The mobility of these hardware ships presents some interesting directions for future research, but for the time being such ships must be considered immobile. Even some ships which are programs may be immobile. Ships need to be hardware independent to be truly useful. If not, they could not move between machines and there would be no way, for instance, to share files between different machine architectures. A likely implementation would then be to define an interpreted intermediate language that would be used to represent ship code and data[1]. In non-CPU intensive programs, interpretive code would not be a major problem. But for others it would be impossibly slow. Those programs which require speed could be compiled into native code. The difficulty of moving such ships between different architectures would lead them to be considered immobile. However, the immobile ships can still use interpreted code ships where possible, thereby lending a reasonable but not

[1]Heterogeneous native-code implementations need not be ruled out, but the implementation effort would be much greater.

total degree of mobility. Compiling intermediate code is another technique that could be applied. All the code of a ship could be compiled upon reaching its destination. This would increase the cost of a move, but the extra execution speed could offset the extra cost. Compiling the intermediate code as it executes is another possibility. This would minimize the amount of compilation done while still yielding a gain in execution speed if any significant amount of code is executed multiple times.

The primitives provided implement the Ship Model but do not ensure the integrity of ship's data. Ships that allow arbitrary code to access their data and code have no assurance that that code will not corrupt their data. While research into the Ship Model will inevitably gain from investigation of the uses of an unrestrictive system, a production version would have to add primitives to address the data integrity problem. A simple but effective solution would be to have ships specify exactly what kinds of ships may enter them. A ship could then guarantee (through its entry restrictions) that only valid code will be allowed to manipulate its data.

The Ship Model has a very general form of inter-ship communication. It is up to a high level language that uses the Ship Model to define what semantics it provides and how they are implemented. For illustrative purposes, we will show how a Remote Procedure Call (RPC)[2] call can be effected. The caller can create a new ship (called a *carrier*) within itself that contains the parameters, entry point and ID of the callee in its data space. The carrier ship's code will obtain the ID of the caller and move to the callee. The thread that initiated the call is suspended. Once the carrier is within the callee, it can bind the entry point to the code, set up the parameters appropriately and execute the procedure call. When the call is complete, the carrier can move back to the caller, place returned data where it is expected, destroy the carrier ship and continue the execution of the caller.

A production version of the Ship Model using RPC as its inter-ship communication method could have a straightforward rule for ship entry restrictions: only ships of the type carrier are allowed to enter other ships. Ships would effectively have RPC as their communication mechanism and would otherwise not be able to access the data and code of other ships. The ships that represent physical hosts would need to break the rule so that ship mobility between hosts would not be restricted. However, these host ships could use the same mobility restriction mechanism to implement user level access control. Additions for security would have to be quite flexible. It should be noted that the mobility restriction technique can lead to a performance improvement. If a ship is only willing to let certain kinds of ship enter, it could store the code for those ships in itself. Only the data of the entering ship need be moved into the ship's address space. In the case of RPC, the only data moved would be the procedure parameters and return values.

Chapter 4

Implementation

A prototype of the Ship Model was implemented in order to test the benefits of the model. It is written in ANSI C and runs under SunOS on both SPARC and 68000 based Sun workstations. The code is quite straightforward and will run without modification on any modern UNIX workstation. The implementation consists of one or two *Ship systems* which provide the ship and thread abstraction, a simple name service and the ability to move ships between Ship systems. Each Ship system is a UNIX process. A single Ship system is useful for simple tests and provides a more tenable debugging environment for developing application software. Two Ship systems are used to run the real tests where ships move across a network represented as a TCP/IP connection. Two Ship systems can also communicate over a dialup serial line by each connecting to a serial line multiplexor.

The code of each ship is represented in terms of a virtual machine that has 14 general purpose 32 bit registers along with a stack pointer and a program counter. The instructions are similar to those found in traditional microprocessors. Ship model primitives are implemented via a system call mechanism which halts the virtual machine interpreter and runs some system code to effect the primitive. The application programmer uses the assembler and works at the level of this virtual machine as high-level language compilers are not available. However, there is a set of perl[1][14] subroutines which somewhat mitigate the task of writing applications.

Files, directories, keyboards and displays are represented by pseudo-ships. These

[1]An interpreted multi-purpose language.

pseudo-ships are immobile and are handled by each Ship system. At the application level they appear to be no different from ordinary ships and are accessed by exactly the same mechanism. The name service of each Ship system (accessible via system calls) is used to locate these pseudo-ships. While a real Ship Model implementation would have permanent identifiers for every pseudo-ship, this prototype actually creates pseudo-ships on the fly based on name server lookups. A real implementation would also determine the actual location of a pseudo-ship, but this prototype uses a simple naming convention (a "*" prefix) to indicate if a pseudo-ship is remote.

Ship movement with a single Ship system only requires some updating of internal data structures. Movement between systems causes writing of ship code and data along with the context of any threads involved. Ship systems also transmit name lookups between themselves. A special optimization was added that reduces communication bandwidth by caching code segments of ships. Once a ship has moved to each Ship system, its code will not be transmitted on subsequent moves as it is already available. After one ship has moved into another a thread can copy data between the two ships using a few special virtual machine instructions. Most of the instructions operate on the data portion of the ship where the thread is executing. A few load and store instructions manipulate data in a different area which is set to point to the ship's data where the thread most recently executed. In this way data can be moved between ships. This also provides a notion similar to the containment hierarchy of the Ship Model, but more rigidly enforced. Movement of ships which contain other ships is not allowed.

The virtual machine interpreter includes a single-stepping debugger for ship code. With it the application programmer can set breakpoints, display register values, examine ship memory and control the execution of a ship. The interpreter itself is generated from a table describing the virtual assembly language. Even this descriptive table is not created directly but generated by a program which handles the details of opcode assignment and instruction decoding and encoding hints. The opcode table is used by two different interpreter generators, both creating a C language based interpreter but with one making much more aggressive use of switch statements for faster decoding. The opcode table is also used by the assembler; this helps insure that inconsistencies between the interpreter and assembler are minimal. This organization also makes it fairly straightforward to change the virtual machine language, a feature which was utilized modestly during development. A few criteria were placed on the choice of virtual machine instructions. First, they should be fairly easy for a human to use since there would be no compiler. Second, any constructions which would unduly slow down an interpreter should be avoided. Third, the machine should look reasonably conventional so that a compiler could be easily written that generates code for it. Also, it is hoped that translation of the virtual machine code into native machine code would be easy, automatic and produce efficient code. The virtual machine has no flag register to address the second consideration. Setting flags like "last result was equal to zero" is expensive in software and very often the results are not used. Conditions are instead handled by branching on immediate register tests. The third condition was followed to make the system as realistic as possible. One possible path for an improved system would be one that uses a compiler that generates intermediate code. The system could either interpret the intermediate code directly or compile it down to native code for higher speed. By choosing a realistic virtual machine, the results can show the possibilities of a more advanced system.

Ship, thread, and memory management are provided by a spartan but usable set of system calls. Ships and threads can be created and destroyed. A ship's data segment can be expanded and contracted. A ship can also specify that notice of its destruction be sent to another ship. This is accomplished by specifying a ship and entry point to which the system will send a so-called *ghost ship* when a ship is destroyed. The prototype also implements the lock, unlock and block synchronization primitives.

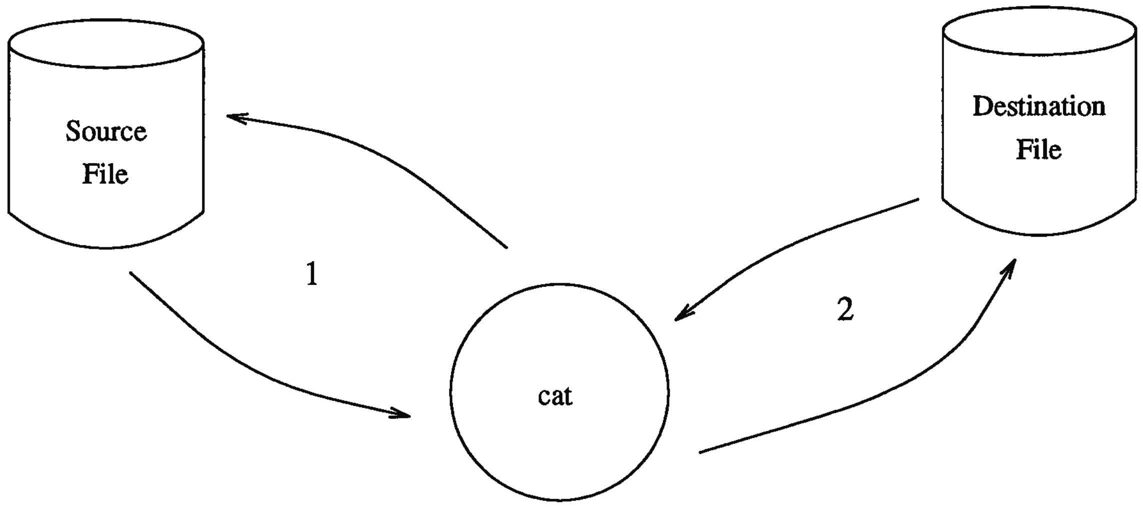
Chapter 5

Sample Applications

This chapter describes some of the applications written for the prototype implementation and their performance. By themselves the applications are of interest as they show how the Ship Model can be used. The performance analysis will show that the Ship Model can realize gains both in high-speed and low-speed network settings. Three applications will be presented. The first is a file copying program which creates a copy of a given file. Three different implementations of that program (`cat`, `cat2` and `copy`) were written and studied. The second is a directory listing application (`ls`) which displays all the files in a given directory. The third is a file browsing application (`more`) which presents a file to the user waiting for input before proceeded to the next page.

Two different sets of hardware were used for timing experiments. A pair of ethernet connected Sparcstation 2 class machines with local SCSI disks were used for the tests in the high-speed networking setting while a Sparcstation 2 and a Sun 3/50 were used for the low-speed network tests. Their connection was through a 9600 baud dialup port. The 3/50 had a modem directly connected to one of its serial ports while the Sparcstation 2 communicated with its modem over ethernet via an Annex III terminal server. As was mentioned previously, the Ship systems on both ends connected to a serial line multiplexing process which did the actual communication with the serial port.

Timing of UNIX commands was done with a simple timing program which times the running of a sub-process. Ship system timing was done automatically. When the first thread starts running the system records the time. Whenever no threads are ready to



`cat` copies the file by moving to the source file (1) and reading a 1024 byte block of data. It then moves to the destination file (2) and writes the data. This continues until the entire file has been copied.

Figure 5.1: The simple copy program, `cat`.

run, the system records a potential finish time. When the Ship system is killed it prints the elapsed time and possibly a warning message if the system was killed while things were still happening.

The first version of the file copying program (`cat`) is very simple. It consists of a single ship which looks up the source and destination files and then moves between the two files copying a 1024 byte block of data each time. See figure 5.1 for an illustration of its workings. Notice that when copying a file within a single Ship system `cat` operates in the same way as a UNIX file copying program (see figure 5.2). When copying between two different Ship systems `cat` effectively implements a stop-and-wait file transfer protocol.

`cat` has a rather large flaw that is corrected in `cat2`, the second version of the file copying program. In the first version, the data size of the ship was never changed. An entire buffer full of data is transmitted on the return trip from the destination file back to the source file. `cat2` shrinks its data size down to a minimum before moving back into the source file ship thereby saving an unnecessary transfer of data.

```
#include <stdio.h>
#include <sys/time.h>

main(argc, argv)
int argc;
char* argv[];
{
    FILE* src, *dst;
    int n;
    char buf[1024];
    struct timeval t0, t1;
    int sec_diff, usec_diff;

    if (argc != 3) { printf("usage: filecopy src dst\n"); exit(1); }

    gettimeofday(&t0, (struct timezone*)0);
    src = fopen(argv[1], "r");
    dst = fopen(argv[2], "w");
    while (n = fread(buf, 1, 1024, src))
        fwrite(buf, 1, n, dst);
    fclose(src);
    fclose(dst);
    gettimeofday(&t1, (struct timezone*)0);

    sec_diff = t1.tv_sec - t0.tv_sec;
    if (t0.tv_usec > t1.tv_usec) {
        usec_diff = 1000000 + t1.tv_usec - t0.tv_usec;
        sec_diff--;
    }
    else
        usec_diff = t1.tv_usec - t0.tv_usec;

    printf("elapsed time: %d.%06d seconds.\n", sec_diff, usec_diff);
    printf("\nt0: %d %06d\nt1: %d %06d\n",
        t0.tv_sec, t0.tv_usec, t1.tv_sec, t1.tv_usec);
    exit(0);
}
```

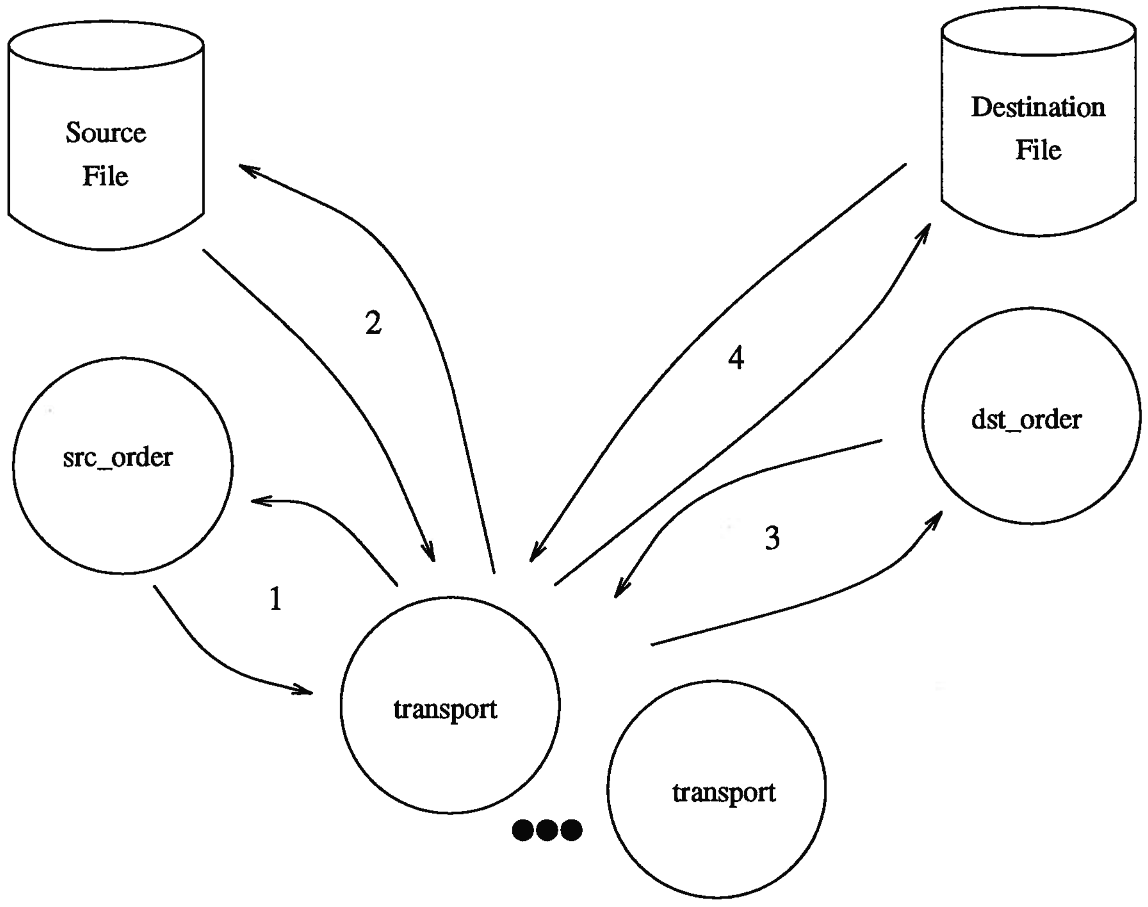
Figure 5.2: A typical UNIX program which copies a file.

Program	A to A from A	A to A from B	A to B from B	A to B from A
cp	0.64			
filecopy	2.6	8.0	2.8	7.6
cat	2.7	3.3	12.8	12.8
cat2	2.9	3.6	13.0	13.0
copy 1	3.7	3.8	12.1	12.1
copy 2	3.8	4.1	8.5	8.5
copy 3	3.6	4.2	8.4	8.4

Table 5.1: Times in seconds to copy a one megabyte file between two Sparcstation 2 class machines A and B.

Despite the simple optimization, `cat2` is still a stop-and-wait protocol. Figure 5.3 shows a third version of the file copying program which uses multiple cooperating ships. By using more than one ship to transport the file data from source to destination, `copyn` effects a windowed file transfer protocol. The `n` represents the number of transport ships used. The multiple *transport* ships pass through the *src_order* and *dst_order* ships before entering the source file and destination file ships so that the file data is read and written in proper order. The src_order ship assigns sequence numbers to each transport ship that enters it and makes sure that only one transport is reading the file at a time by locking out any other transport until the current one has finished. The dst_order must impose a stricter ordering based on the sequence numbers assigned to each transport. Each transport arriving will run within the dst_order ship. If it does not have the correct sequence number, it will voluntarily block. Otherwise, it continues on to the destination file, writes its data, updates the sequence number and unlocks the dst_order ship. Any transports blocked waiting to get into the dst_order ship will be unblocked by the unlock and proceed as usual to check the sequence number.

Table 5.1 summarizes the measurements performed on several versions of the file copying program running in the high-speed network setting. The tests can be broken



After the *src_order* and *dst_order* ships have been located near the source and destination files respectively, one or more *transport* ships work to copy the file. Each one first moves into the *src_order* ship (1) which guarantees the only one *transport* will read from the file at a time. The *transport* ship then moves into the source file (2) to read a 1024 byte block of data. It then moves to the *dst_order* ship (3) which ensures that the transports write their data blocks in the correct order based on sequence numbers generated by the *src_order* ship. Finally the *transport* ship moves into the destination file and writes the data. The cycle is repeated from step one until the *transport* ships have copied the entire file.

Figure 5.3: copy, an efficient file copying program.

down into four general cases. The first is when the copy of the file will ultimately reside on the same disk and all operations are executed locally. The second is when the file is to be copied onto the same disk but the operations are initiated remotely. The third and fourth cases are copying a file between disks on different machines.

Completely local tests show the baseline performance of each method. As expected the fastest times are from the native UNIX cp program and the C `filecopy` program. It is strange that cp is so much faster when copying files on a local disk, but this is because it uses SunOS' mmap() routine to read and write the files instead of the traditional UNIX read() and write() routines. Presumably mmap() can avoid some buffer copying and utilize kernel level parallelism to gain such an increase in speed. The Ship system times lag slightly behind `filecopy` with the more complex versions paying an extra small time penalty because of the extra operations involved. Table 5.2 shows the cost of three basic

Operation	Sparc 2	3/50
instructions	260,000/s	45,000/s
move operations	20,000/s	2,000/s
data area resizes	14,000/s	1,200/s

Table 5.2: Cost of some basic ship operations in operations per second.

ship operations. An instruction is one virtual machine instruction. A move operation represents the cost of moving one ship into another in a single Ship system. A data area resize is when an individual ship either increases or decreases the data space available to it. Table 5.2 can be used to determine the cost of the extra operations in `cat2` and `copy`. They both require two data resize operations per block of file data copied which will impose approximately a 0.14 second penalty. This is a relatively small increase and does not show up well in the `cat2` timings. However, where `cat` only does four moves per block of data, `copy` does twelve. The extra 0.4 seconds required for these moves has a noticable effect on the timings.

Local copying that is initiated remotely gives the Ship system the best advantage. After the initial setup of locating the files and moving a few ships, the ship programs will execute exactly the same as their purely local invocations. This setup is reflected in their times as each requires just a bit longer than the local versions. The time for copying increases slightly with the number of transport ships used in the `copy` program. As it stands the Ship system does not provide fair scheduling and only checks its network connection when no threads are running. A transport moving between two files in a single Ship system will run without blocking. Therefore the first transport ship across in the `copy` program starves the others out until the copy is finished causing a small extra delay when the rest arrive and promptly exit.

Copying files between machines differentiates the ship programs and reveals an asymmetry in NFS [10]. While `cat2` could have an advantage over `cat`, none is shown since the extra cost of data on the return trip is relatively small. `copy` improves with the number of transport ships used since its effectively windowed protocol now has an advantage over the stop-and-wait nature of `cat` and `cat2`. When the `filecopy` program runs on the machine storing the destination file it goes much faster than when it runs on the machine storing the source file. This is because NFS requires that write operations not be acknowledged until the data is written to disk so writing is much slower than reading.

The results of some file copy tests over the slow speed line are shown in table 5.3. The test files used were compressed versions of ordinary files to minimize any artificial effects due to the modem's data compression feature. Here the ship implementations are well differentiated because the line speed dominates the transfer times. `cat2` is considerably faster than `cat` because of a greatly reduced return time due to its data segment shrinking optimization. When given more than a single transport ship, `copy` outperforms the others because of its effective windowing protocol. The startup cost of

Program	Copy time for 20635 bytes	Copy time for 71459 bytes
cat	56.5 (365 char/s)	
cat2	35.4 (582 char/s)	
copy 1	36.1 (572 char/s)	118.1 (605 char/s)
copy 2	24.6 (835 char/s)	76 (940 char/s)
copy 3	24.8 (838 char/s)	75.4 (947 char/s)
kermit (1K packets)	34 (607 char/s)	114 (627 char/s)
kermit (windowed)	26 (794 char/s)	85 (840 char/s)
zmodem	21 (982 char/s)	66 (1083 char/s)
raw	18.6 (1109 char/s)	63 (1134 char/s)

Table 5.3: File copy times in seconds over a 9600 baud serial link between a Sparcstation 2 and a Sun 3/50.

sending the code of each ship the first time can be seen in that the large file gives slightly greater throughput due to the amortization of the startup cost. It is interesting to note the similar performance of the stop-and-wait `cat2` ship and the stop-and-wait version of kermit. A similar similarity is seen between the "windowed" `copy` and the windowed version of kermit. While competitive with kermit, the best ship file copier cannot keep up with the very efficient zmodem protocol. This difference is mainly due to the 136 byte overhead associated with each transport ship. This effectively increases the file size by 13 percent. With this overhead factored in we see that the `copy` program is using 1070 characters per second or almost all of the serial line's bandwidth. Since much of the thread context transmitted is unnecessary (the virtual machine registers are transmitted but they are also typically saved on the stack), a system hint could reduce the overhead to 80 bytes. Simple compression techniques as used on serial line TCP/IP packets [5, 9] could lower this overhead even more. If both techniques were applied, the overhead could be dropped to 20 bytes or less giving an estimated throughput of 1050 characters per second.

Table 5.4 gives the results of initiating a file copy on the 3/50 which copies a 70K file

Program	Time to copy a 71459 byte file (seconds)
cat	2.3
cat2	1.8
copy 1	2.2
copy 2	2.3
copy 3	2.3

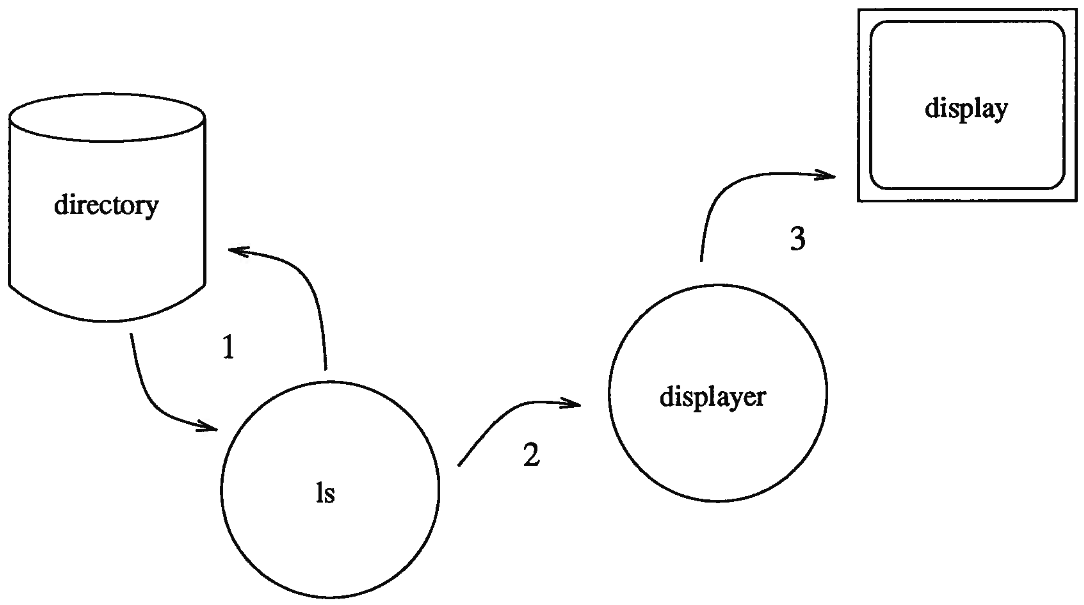
Table 5.4: Times to copy a 71459 byte file on a Sparcstation 2 when initiated from a Sun 3/50 over a 9600 baud line.

	50 files	100 files	200 files
UNIX ls	0.49	0.78	1.3
ship ls	0.15	0.34	0.74

Table 5.5: Times in seconds to list variously sized directories.

on a Sparcstation 2 (both the source and destination file are on a single machine). The copying time is largely dominated by the time to get the ships to the remote site. While the result is completely as expected, it serves to illustrate that the general mechanism of the Ship Model can achieve tremendous results over location-rigid programs. This result is not comparable with either kermit or zmodem since those programs are designed to copy files between machines; they are the tools of a loosely coupled distributed system. A remote file system such as NFS would score very poorly on this test. Even ignoring protocol overhead, if the 3/50 used NFS to copy the file it would end up transferring the file to the 3/50 and then back to the Sparcstation 2 requiring at least 63 seconds for each transfer giving a lower bound of 126 seconds.

A directory listing application called `ls` was created to show another advantage of the Ship Model. Under UNIX, reading a directory is similar in cost to reading a small file. But a directory listing with extra information such as file type requires not only reading the directory but also performing an operation for each file to get the extra information.

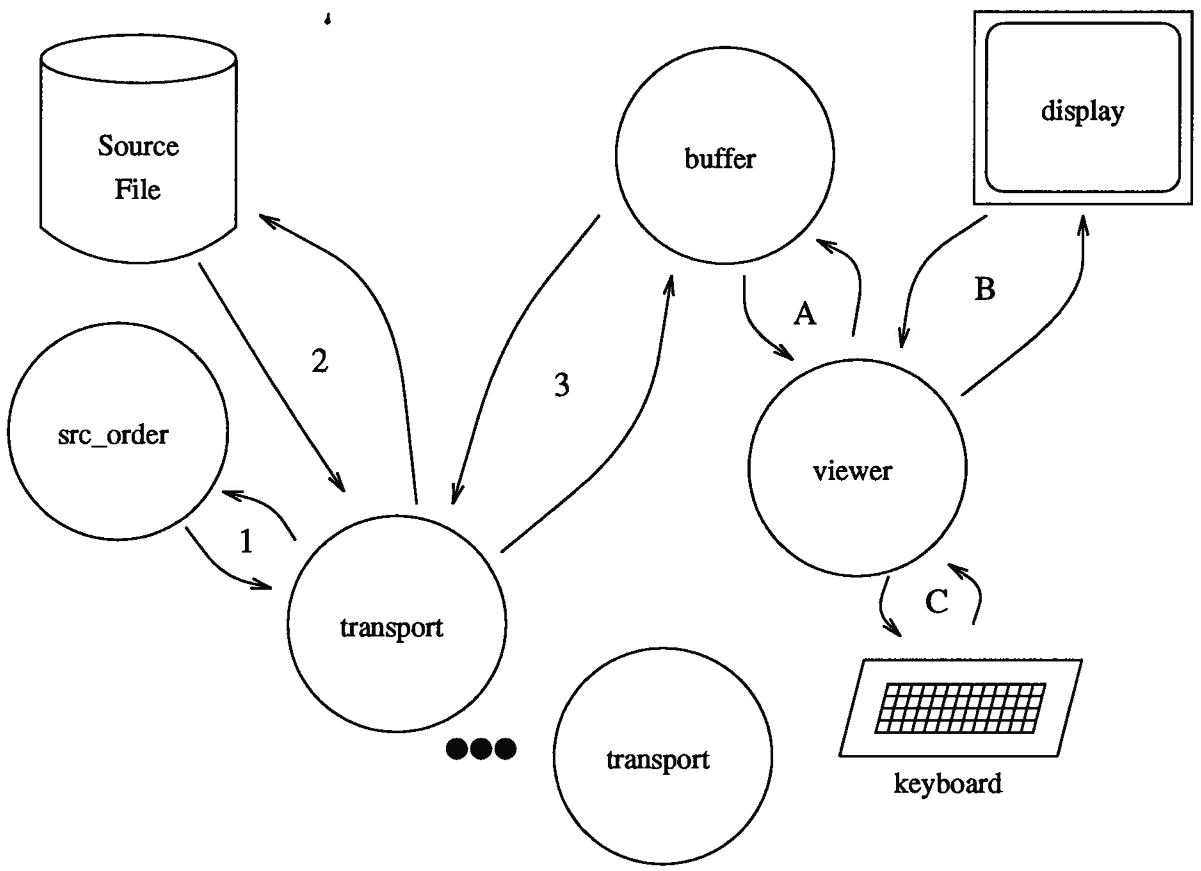


The *ls* ship starts by moving to the directory (1) and reading all the directory entries. It then creates a *displayer* ship (2) which contains the directory information and moves to the display (3) and prints the information for the user.

Figure 5.4: `ls`, a directory listing program.

Table 5.5 compares UNIX ls against the ship ls for directories with various numbers of files. Notice that each time increases approximately linearly with the number of files but that the ship ls is consistently faster. Figure 5.4 shows the operation of `ls`. It gains its advantage by performing all the per-file operations on the file server rather than doing each one separately.

An interactive page browsing program was the final application tested. As shown in figure 5.5, it is very similar in construction to the copy program. The transport ships move the file data to a buffer ship which holds the data for the viewer ship which actually moves the data from the buffer to the display. The advantage of this `more` is that the file is transferred while the user is viewing the file. As long as the user reads more slowly than the file data is transferred each new page of data is presented quickly, more quickly than the low speed line can deliver the data. In fact the user will typically have a hard time keeping up with even a slow serial line since human reading speed is quite slow. However, it should be noted that sometimes a human will outpace even high speed pagers when scanning for some particular bit of text. A simple calculation shows that executing the interpreted code of the viewer ship can outpace the slow-line data rate. The viewer code scans the input data so that it can count lines displayed and pause after a screen full. The scanning imposes an overhead of eleven instructions per character. Even the 3/50 can outpace the 1100 character per second speed by being able to process 4100 characters per second. A sparc 2 class machine does much better. These numbers can quite safely be viewed as lower bounds since there is room for improvement in the C interpreter and an assembly language version would likely run at least twice as fast.



Two subsystems work concurrently to present a file to the user. One or more *transport* ships order their file reads by moving to *src_order* (1) which gives them a sequence number. They then proceed to read a 1024 byte chunk of the file (2) and then move to the *buffer* ship (3) where they copy the file data in order based on their sequence numbers into the *buffer* ship. The *transport* ships repeat this process until the *buffer* ship contains the entire file. Meanwhile, a *viewer* ship moves into the *buffer* ship (A), reads a line and moves to the display (B) until a full page has been presented. It then moves to the keyboard (C) and waits for a key to be pressed before proceeding from the start. During setup the *src_order* ship is located near the source file and the *buffer* ship is located near the display so that most of the ship moves are entirely local.

Figure 5.5: `more`, a file browsing program.

Chapter 6

Comparison with Related Work

This chapter presents several systems that address the object location problem and shows that the Ship Model is a general model capable of satisfying the goals of each of the systems. Thus the chapter is intended to show the power and generality of the Ship Model as well as some useful applications.

6.1 Emerald

Emerald is a distributed operating system and language developed at the University of Washington [3]. Emerald provides a single object model and explicit support for object mobility [6]. It is designed to operate in a local area network with a modest number of homogeneous hosts. Although Emerald allows different object implementations, they are all defined and accessed with a single, uniform set of semantics. Emerald objects optionally contain a process and multiple threads of control are synchronized with monitors.

The primitives for object mobility are: locate X, move X to Y, fix X at Y, unfix X and refix X at Y. Any object may be used to specify a location. The location indicated by an object is simply the location of that object. Special objects called node objects are used to specify a particular location by having them permanently attached to a location.

Emerald uses remote and local procedure calls for communication between objects. All parameters are object references. Emerald takes advantage of its fine-grained object mobility to make the RPC efficient by allowing objects to migrate to the site of a remote operation. The migrated objects may return when the call is finished (call by visit) or remain at the remote site (call by move).

Object mobility presents a difficulty for RPC in Emerald. A local invocation can turn into a remote invocation if the invoking object or the invoked object moves to a different location. Emerald handles this by moving activation records with the call.

The move primitives of Emerald and the Ship Model differ in that the *move* primitive is treated as a hint by Emerald and no move is necessarily done. The Ship Model does not have any primitives like *fix* because they are easily implemented in terms of the Ship Model (see section 3.1). There is a careful distinction to make between Emerald's and the Ship Model's notion of location. In the Ship Model, a ship *is* a location while in Emerald an object identifier evaluates to its location, but is not actually a location itself.

Emerald and the Ship Model use very different forms of inter-object and inter-ship communication. Emerald allows objects to communicate via procedure calls. The procedure calls may be either local or remote and the semantics of both are the same. The Ship Model uses ship mobility as its communication method. Since ship mobility can simulate remote procedure calls, the Ship Model has the advantage of a more general communication mechanism – other forms of communication can be simulated including techniques not found in other systems. With mobile objects to consider, the procedure call mechanism in Emerald becomes more complicated. An implementation must be prepared to translate local procedure calls into remote ones. This results in the state of a process being spread across many machines. In the Ship Model, processes (threads of control) are always contained on a single machine which makes process control schemes easy to implement.

Emerald uses object mobility to enhance RPC communication between objects in a distributed system. The Ship Model has mobility as an end instead of a means. These different foci constitute the main difference between the two approaches.

6.2 NCL

The Network Command Language (NCL) [4] is part of a prototype heterogeneous distributed system developed at DEC. The main goal of this system was to link together the diverse products offered by DEC. To that end, a client/server model was chosen. However, there remained some difficulty as to how to implement the system. The problem was a general one of server interface design. The servers had to provide an interface for a number of different clients based on the original interface that the client's old system provided. One possible solution is to simply provide all the different types of client interfaces. But the resulting server would be unwieldy and complicated to implement. An elegant solution is to make the server implement a "lowest common denominator" interface made of primitives that can be combined to create the desired client interface. This approach makes the servers simpler, but a single logical client operation often results in a number of actual server requests. Efficiency suffers as a result.

NCL was created to solve the server design problem. It is a lisp-like command language implemented by all servers on the network. When a client makes a request, it formulates it in terms of NCL and sends the NCL expression to the server. The server evaluates the NCL expression and returns the results to the client. The expression may execute many different server requests thereby effecting many remote operations with only a single remote request. By using NCL, a server may implement a simple interface without penalty to those clients who make many requests to fill a single logical request.

A client/server distributed system could use ships in the same way it might use NCL. The client would create a ship which moves to the server, interacts with the server locally and then moves back to the client to return its results. This method would provide the advantages of NCL, namely doing many remote operations locally. The Ship model would enhance this method of operation since a ship has its own thread of control. In NCL, the expression is evaluated by the server. Servers must be careful that NCL expressions do not use up too many resources since their responsiveness depends on dealing with client requests quickly. A ship interacting locally with a server does not require the server to monitor the execution of the ship. Indeed, the long term interactions give ships extra opportunity to exploit locality. NCL could not implement the file copying example since a large file would cause the copying expression to be terminated by the server.

6.3 NeWS

Sun Microsystem's NeWS (Network Extensible Windowing System) [12] bears some similarities to the Ship Model. NeWS uses a client/server Model. The servers mediate access to the display, keyboard and mouse of a workstation. The clients are the applications programs which wish to use bitmapped displays for output and keyboards and mice for user input. What makes NeWS interesting is that the clients use a PostScript-like language to communicate their requests to the server. In essence, the server is an interpreter – it evaluates expressions and returns the results to the clients. The expressions can draw images on the screen or read keyboard and mouse input. The expressions can also define procedures. This is what makes NeWS extensible. The idea is that a client will define operations appropriate to its function and will only need to send the names of those operations instead of the entire operation each time.

NeWS gains in many ways from its approach. The PostScript-like language serves also as an external data representation. Extensibility means that the server only needs to implement basic graphic and input operations. Clients who need to draw a square can define a new primitive based on line drawing operations. Communication between client and server is reduced since local code can handle many functions by itself without remote client intervention. In fact, there are many NeWS programs which are completely local to the server (i.e., are just a PostScript program). The Ship Model strives for many of the same goals as NeWS. It too can exploit locality of reference through ship mobility. It can also reduce the complexity of server design since locality can be exploited. While extensibility is not a direct part of the Ship Model, an implementation could support the concept. In order to do this, the implementation would move a ship in two parts. First, the data of the ship would be moved. Then, if the code of the ship was not already available on the remote machine, the code would be moved. The code may be available because of caching or it may be on a local disk on the remote machine.

Even though both the Ship Model and NeWS both seek to use locality of reference to their advantage, it is unfair to compare them overall. NeWS is not intended to be more than a windowing system while the Ship Model is designed to be a general purpose system. It is interesting to note that NeWS programmers face some problems because of their flexible environment. While a file may be available on the client machine, it may not be on the machine running the NeWS server. A program to display a file must be careful to read the file on the client machine and send the information to the server. But the file location problem is a general one and the solution should not have to be programmed for each client. Also, if the client and server are on the same machine, data is travelling unnecessarily through the client. With the Ship Model in place, a ship could get the file name, move to the machine where the file is and transmit the file back to the server. Not only does it reduce client complexity, it also makes for an optimal communication path.

6.4 sam

sam the text editor was designed to run on a Blit terminal [8]. The Blit is a simple graphics terminal that allows code to be downloaded and run. Because of this, **sam** was designed as two processes. One runs on the main host while the other runs on the Blit. The process on the Blit handles keyboard and mouse input and screen updates. The process on the host does all the changes to the file. To reduce the amount of communication between the host side and the Blit side, the Blit side maintains a data structure which keeps track of the information it knows about a file. When a part of the file must be displayed, the Blit side looks at this data structure and uses the information there whenever possible. If it does not have the information to update the screen, it then interacts with the host side and adds the new data to its data structure. In this way the Blit side provides good user interaction since much of the editing process displays good locality of reference. While `sam` has a number of features which make it desirable as an editor, one of its primary benefits is its ability to provide good interaction over a relatively slow serial line. The Ship Model is designed to exploit locality of reference and an editor written using the model has the potential to be as useful as `sam`.

`sam` can be run in a slightly different three process configuration. In this setup, the Blit side remains the same and the host side is put on a file server. The third process is on the machine to which the Blit is physically connected and simply passes data between the two original parts. It has been noted that this can lead to superior performance since accessing the files directly on the server is faster and often the file server is a much more powerful machine. Although `sam` initially sought to increase the usefulness of editing over slow serial lines, a side benefit was to make a faster editor over high speed networks. The Ship Model is also primarily designed to make slow lines more useful, but `sam` illustrates that there are potential benefits in applying low speed techniques to high speed networks as well.

The implementation of `sam` differs from the Ship Model because no code (except the initial Blit program download) is dynamically moved between the host and the Blit. The host/Blit interaction protocol in `sam` is specially defined and is not used for other purposes.

Chapter 7

Conclusions

The Ship Model was developed to address the problem of resource location in a distributed system. While remote access to a resource is a necessary part of a distributed operating system, it is undesirable or even acceptable when communication with that remote resource is slow because of substantial bandwidth requirements or low speed communication links. The Ship Model addresses this problem by allowing programs to move code and data encapsulated in ships to remote sites. The mechanism has two effects on communication requirements. First, it allows a program to use the most direct paths of communication. This advantage was shown in the file copying application. The file data being copied was guaranteed to be sent no more than once over the network. Moreover, in the case where the file was to be copied to the same machine the file data was not sent over the network at all. A typical file server arrangement such as NFS may require the file data to cross the network twice for a single copy in some cases. Second, a program can coalesce multiple remote operations into a single ship move. This translates the cost of n remote operations into the cost of a single remote operation and n local operations. While network usage will probably be reduced because the overhead of $n-1$ remote operations is eliminated, the performance advantage is more significant since $n-1$ round trips are eliminated. The gains from using the Ship Model in this way were seen in the directory listing application.

A nice side effect of the Ship Model is the simplification of device interfaces. Take, for example, a terminal. There are only two necessary interface routines: read some characters and write some characters. Extending the interface for remote access will require that more routines be added lest response be lowered and network bandwidth be wasted on single characters going back and forth. Since the Ship Model allows the programmer to send a ship to the terminal, the terminal interface can remain simple. The system interface to the terminal is kept simple and the decision on what user input will generate network traffic is left to the application. The prototype system demonstrates this property in that its terminal interface is broken down even further into keyboard and display interfaces. The keyboard has a single routine which returns a key while the display has a single routine which displays characters.

There are some weaknesses in the Ship Model. Experiments with the prototype showed that the Ship Model is slower to copy files when the file is moved over a high speed network. It is also slightly slower than some file transfer programs when operating over a low speed serial line. In part these problems can be explained by limitations of the prototype. Since the prototype is implemented on top of UNIX, it is limited to utilizing the intermachine communication provided by UNIX which imposes its own overhead and inefficiencies on the prototype. In the case of file copying over a high speed network, a more finely tuned Ship system or one implemented independently of UNIX should be able to equal or surpass UNIX's performance. In comparing the prototype against other serial line file transfer protocols it should be noted that the Ship Model provides a general mechanism while those protocols are designed for the sole purpose of transferring files from one system to another. Despite this, the performance gap is not great and could be closed with some extra work on reducing the communication overhead.

The Ship Model's ability to work across heterogeneous machine architectures presents some difficulties. The solution taken by the prototype – representing ship code in terms of a virtual machine – was chosen for its simplicity. Fortunately, the overhead of interpreting code has little effect on many applications as demonstrated by the file copying and directory listing programs. The file browser faced a potential bottleneck because of the interpreter but the interpreter's speed was still several times faster than that required for the job. However, there will be applications where the interpreter is too limiting. Fortunately there are other ways to approach the heterogeneity problem. An intermediate code representation with compilation to native code would produce faster running ships. Or multiple versions of a ship's code for each machine type in the system could be tracked with the ship's data represented in some machine independent form.

The prototype provides a minimal system in which to explore the Ship Model. It can not address the question of what it is really like to use the model at a high level. The main concern is that carefully structuring a program to minimize network communication will overwhelm the programmer. This structuring was not a problem with the prototype since other concerns such as writing assembly-like code were much more difficult to deal with. Only experience with a high-level language for the Ship Model will answer this question, but some extrapolations can be made. At the design level the Ship Model will be of some help since the programmer can at least recognize when network communication is taking place. Any move represents possible network use and nothing else does. It can also be expected that low communication systems can be encapsulated into library ships for ease of use and re-use. A ship for reading a line at a time from a terminal is a likely example, but even more complex mechanisms such as the transport ship system used in the copy program could be made into library ships.

There are three main areas for future research: the basic mechanics of an implementation, creating a usable system and creating a high level language. Any implementation must deal with heterogenous machine architectures. Devising a scheme to speed up the code running speed of a ship is a difficult but worthwhile goal. An implementation which can equal or beat UNIX in copying files over a high-speed network would be an important step in demonstrating the viability of the Ship Model. A system that could be used on a day to day basis would be valuable. The prototype was set up only to run one-shot experiments. Using the system on a long term basis to do real work will show that the Ship Model can provide overall benefits. This will be especially true when operating over a serial line – the interaction will be better than terminal emulators or even remote X servers. How the Ship Model would be used by a high level language or in the design of a high level language remains unanswered. It will be interesting to see if the potential design complexity can be managed while still producing communication bandwidth stingy programs.

Appendix A

The prototype virtual machine

The prototype uses a virtual machine with sixteen 32 bit general purpose registers named r_0 through r_{15}. r_{15} is used as the program counter while r_{14} is used as the stack pointer. The registers can be accessed in 8, 16 and 32 bits at a time and most instructions support these accesses as byte, word and longword modes. The virtual machine can access two separate data segments. Most instructions only use the primary data segment, but some instructions manipulate data in the secondary segment. The second data segment is used to allow data to be copied between two different ships.

Here is a list of instructions. Unless otherwise specified, every instruction can use the virtual machine registers in byte, word or longword mode. That is, the instruction will operate on the lower 8, 16 or 32 bits of the registers involved.

add r_i,n: The immediate value n is added to register r_i.

add r_i,r_j: Register r_j is added to register r_i.

and r_i,n: Register r_i is logically anded with the immediate value n.

and r_i,r_j: Register r_i is logically anded with register r_j.

div r_i,n: Register r_i is divided by the immediate value n.

div r_i,r_j: Register r_i is divided by register r_j.

mul r_i,n: Register r_i is multiplied by the immediate value n.

mul r_i,r_j: Register r_i is multiplied by register r_j.

or r_i,n: Register r_i is logically ored with the immediate value n.

or r_i,r_j: Register r_i is logically ored with register r_j.

sub r_i,n: The immediate value n is subtracted from register r_i.

sub r_i,r_j: Register r_j is subtracted from register r_i.

xor r_i,n: Register r_i is exclusive ored with the immediate value n.

xor r_i,r_j: Register r_i is exclusive ored with register r_j.

cpl r_i: The contents of register r_i are twos complemented.

not r_i: The contents of register r_i are ones complemented.

shl r_i, r_j: The contents of register r_i are shifted left by the number of bits specified in register r_j.

shl r_i, n: The contents of register r_i are shifted left by n bits.

shr r_i, r_j: The contents of register r_i are shifted right by the number of bits specified in register r_j.

shr r_i, n: The contents of register r_i are shifted right by n bits.

sys n: System entry point n is called (i.e., break out of the interpreter).

push *regmask*: Register r_i is pushed on the stack if bit i of *regmask* is set. All registers may be pushed with one instruction, but only the longword form of this instruction is available.

pusho *regmask*: Same as push except that the registers are placed in the secondary data segment.

pop *regmask*: Register r_i is popped from the stack if bit i of *regmask* is set. All registers may be popped with one instruction, but only the longword form of this instruction is available.

popo *regmask*: Same as pop except that the registers are retrieved from the secondary data segment.

load r_i,n: Register r_i is loaded with the immediate value n. Note that loading $r_1$5 with a value effects a goto.

load r_i,r_j: Register r_i is loaded with the contents of register r_j.

load r_i,(r_j): Register r_i is loaded with the contents of the primary memory segment location pointed to by r_j.

loado r_i,(r_j): Register r_i is loaded with the contents of the secondary memory segment location pointed to by r_j.

store r_i,(n): Store the contents of register r_i into the primary memory segment location n.

storeo r_i,(n): Store the contents of register r_i into the secondary memory segment location n.

store r_i,(r_j): Store the contents of register r_i into the primary memory segment location pointed to by r_j.

storeo r_i,(r_j): Store the contents of register r_i into the secondary memory segment location pointed to by r_j.

load0 r_i,r_j,r_k: Load register r_j with the contents of register r_k if register r_i is equal to zero. If r_j is $r_1 5$ this is effectively a conditional branch instruction.

loadc r_i,n,r_j,r_k: Load register r_j with the contents of register r_j is the nth bit of register r_i is one.

Appendix B

Source code for the cat2 application

The source code for the applications is actually a perl script which generates the assembly language virtual machine code. The "&foo()" constructs are calls to library routines which handle various tasks such as allocating registers and building stack frames. The code is not fully explained but should still give the reader a feel for what a ship program looks like.

```
#!/cs/local/bin/perl

# Copy "file1" to "file2" as specified by argv[0] and argv[1]
#
# Slightly improved over cat.sp in that we shrink down between moves...

require '../macros.pl';
$thisship = 'cat2';

&import("file");

&main;
&needargs(2);

$bufsize = 1024;

$srcfile = &getreg;
$dstfile = &getreg;
$buf = &getreg;
$len = &getreg;

&get_argv($srcfile, 0);
&get_argv($dstfile, 1);
&locate($srcfile, $srcfile);
&locate($dstfile, $dstfile);
&lastmem($buf);

&label("loop");
&movenear($srcfile);
&load($len, $bufsize);
&sbrk($len);
&move($srcfile, "file.read", $buf, $len, '@', $len);
&if($len, '=', 0, "done");
&move($dstfile, "file.write", $buf, $len, '@', $len);
&sbrk(-$bufsize);
&goto("loop");

&label("done");
&move($dstfile, "file.close", '@');

&finishup;
```

Bibliography

[1] Eric J. Berglund, "An Introduction to the V-System" *IEEE Micro*, Aug. 1986, pp. 35-52

[2] Andrew D. Birrell and Bruce Jay Nelson "Implementing Remote Procedure Calls" *ACM Transactions on Computer Systems*, Vol. 2, No. 1 (Feb. 1984), pp. 40-59.

[3] Andrew Black, Norman Hutchinson, Eric Jul and Henry Levy "Object Structure in the Emerald System" In *Proceedings of the 1986 ACM Symposium on Object Oriented Programs, Systems, Languages and Applications*, Sept. 1986, pp. 78-86.

[4] Joseph R. Falcone, "A Programmable Interface Language for Heterogeneous Distributed Systems" *ACM Transactions on Computer Systems*, Vol. 5, No. 4 (Nov. 1987), pp. 330-351.

[5] V. Jacobson, *Compressing TCP/IP Headers for Low-Speed Serial Links.* Arpanet Working Group Requests for Comment, DDN Network Information Center, SRI International, Menlo Park, CA, Feb. 1990. RFC-1144.

[6] Eric Jul, Henry Levy, Norman Hutchinson and Andrew Black "Fine-Grained Mobility in the Emerald System" *ACM Transactions on Computer Systems*, Vol. 6, No. 1 (Feb. 1988), pp. 109-133.

[7] S. J. Mullender and A. S. Tanenbaum "The Design of a Capability-Based Distributed Operating System" *The Computer Journal*, Vol. 29, No. 4 (Aug. 86), pp. 289-299.

[8] R. Pike. "The Text Editor sam" *Software – Practice and Experience*, Vol. 17, No. 11 (Nov. 1987), pp. 813-845.

[9] W. Simpson, *The Point-to-Point Protocol (PPP) for the Transmission of Multi-protocol Datagrams over Point-to-Point Links.* Arpanet Working Group Requests for Comment, DDN Network Information Center, SRI International, Menlo Park, CA, May 1992. RFC-1331.

[10] R. Sandberg, D. Goldberg, S. Kleiman, D. Wals and B. Lyon "Design and Implementation of the Sun Network Filesystem" Sun Microsystems, Inc. Technical Report, 1985.

[11] Andrew S. Tanenbaum and Robbert Van Renesse "Distributed Operating Systems" *Computing Surveys*, Vol. 17, No. 4, Dec. 1985, pp. 419-470

[12] Technical Reference Manual for NeWS, Sun Microsystems.

[13] Marvin M. Theimer, Keith A. Lantz and David R. Cheriton "Preemptable Remote Execution Facilities for the V-System" *ACM 10th SOSP*, Dec. 1985, pp. 2 - 12.

[14] Larry Wall and Randal L. Schwartz, *Programming Perl.* O'Reilly & Associates, Inc., Sebastopol, California, 1990.

[15] Bruce J. Walker, Gerald J. Popek, Robert English, Charles Kline and Greg Thiel "The LOCUS Distributed Operating System" *ACM SIGOPS Operating Systems Review (Ninth ACM Symposium on Operating Systems Principles)*, Vol. 17, No. 5, (Oct. 1983), pp. 49-70

[16] E. Zayas, "Attacking the Process Migration Bottleneck" *ACM 11th SOSP*, Nov. 1987, pp. 13-24.