Using Events to Implement a Distributed Worm in a Mobile Environment

Submitted to The University of Dublin in

Partial Fulfillment of the Requirements for the

Award of the degree of **Master of Science**.

Robert Ashmore B.Sc.

The University of Dublin,

Department of Computer Science,

Trinity College,

Dublin 2.

Library Declaration

This dissertation is entirely the work of the author, except where otherwise stated, and has not been submitted for a degree at any other University. This dissertation may be copied and lent to others by the University of Dublin.

Robert Ashmore

September, 1999

I dedicate this work to my wife Jeanette and my children Tegan and Scott, for their patience and understanding throughout the duration of this course.

Acknowledgments

I should like to acknowledge the assistance of my supervisor Brendan Tangney, without whose patience and understanding, this work would not have been possible. I would also like to thank Stephen Weber and Simon Dobson for their help in enlightening my path to the finer points of Java, and to Tarlach Baumgarten who introduced me to the Zen of Unix

Robert Ashmore

The University of Dublin

September 1999

Abstract

Robert Ashmore
The University of Dublin, 1999

Supervisor: Mr. Brendan Tangney

A distributed system comprises a number of independent computers linked together by a network, running a set of software components residing on numerous machines, all working towards a common goal.

A worm is a paradigm for distributing certain types of parallel computations across a network. A distinguishing feature of the worm paradigm is its support for a certain degree of fault tolerance. Key to the successful implementation of a worm or any distributed application is the communications paradigm. Another distinguishing feature of a worm is its ability to migrate.

This dissertation focuses on using the worm paradigm in a mobile environment. In particular the dissertation is an examination of the suitability of the event communication paradigm in this environment. The design and implementation of an event service is reported. A novel feature of the event service is its fully distributed nature, and its support for mobility in both consumers and producers of events.

A worm was implemented using the event service, and a number of application programs were written to demonstrate its operation.

Contents

List of Figures

Chapter 1

Introduction

This introduction covers the motivation for using, and the general requirements for developing a worm. The general concept of a worm is reviewed. Many of the services a worm would require to work in a mobile environment are introduced. The introduction sets out the general scope of the worm project, and the constraints under which it it has been developed.

The last decade of the 20th Century has seen great advances in communications. The technology to exploit these advances has opened up many new possibilities for computer scientists working in the field of distributed systems. Two technologies, the Internet and wireless communications, have become major factors influencing the research and development of new distributed algorithms.

The Internet has been with us for many years. Starting in the 1970's as a DARPA sponsored experiment, by the the late 80's, it had grown to encompass much of the academic community. Only in the last decade of the 20th Century with the birth of the World Wide Web, has the the Internet been embraced by the general public, now in technologically advanced societies it is firmly established.

The tools required to maintain this huge network were born in the research labs of distributed systems groups throughout the academic world and industry. As the growth of the Internet has become exponential, problems of limited bandwidth in the underlying communications network's have driven research into overcoming this problem. There are two possibilities that could alleviate the bandwidth problem.

First upgrade the underling networks to carry much greater bandwidths, this is being done, but it has not maintained pace with the growth in Internet users. The second solution is to move some of the processing at present residing on the servers, onto the client's workstations, therefor allowing more efficient use of the current bandwidth.

Wireless technology is only now becoming a possibility as part of the communications info-structure of the Internet, it has proved itself in the telephony arena with the dramatic success of the mobile phone. Extending local area networks and the Internet using wireless technology opens up many new possibilities, and challenges in the field distributed computing.

Add to this environment the huge growth in the deployment of personal computers. Today in offices many employees have computers on their desks. Most of these computers are linked by Local Area Networks, that in turn, may be linked to the Internet. These computers spend much of there life idle, awaiting user input. All these idle workstations make up a resource that has yet to be exploited.

1.1 The Worm

The concept of a worm as the vehicle for a distributed computation is not new. The idea comes from the book, The Shockwave Rider [7] and the first work published on this style of computation was by Shoch and Hupp [42] in 1982.

A worm is a distributed computation made up of many individual segments. Each segments normally resides on a separate machine. One of the defining features of a worm is its ability to provide fault tolerance to its segments. If a segment fails, the worm will automatically replicate the dead segment. The degree of fault tolerance may be effected by the nature of the computation. A worm where all segments are identical, and the segments state, when it fails is not required, can be fully fault tolerant.

During the course of a computation segments are allowed to migrate across machine boundaries. Forced migration allows host machines to reclaim resources used by resident segments. The worm may also require segment migration to allow efficient

use of idle workstations. An important characteristic evolves from this model, if the computation exhibits parallelism the worm paradigm will exploit this, and should complete the computations faster than its single processor equivalent.

1.2 The Communications Model

Communication is key to the implementation of a worm. In nature communications is fundamental to the development of higher forms of life, and in particular human society could not function without effective means of communications.

In a centralized computer process, communications in normally implicit in the call stack of the running program. In a distributed system using technologies such as, Remote Procedure Calls or CORBA, this is also the case. These types of function or method call models are highly coupled and synchronous.

Using a highly coupled and synchronous communications model for a worm that is designed to exist in a mobile an dynamic environment would prove very difficult. There are three general communications models that may be suitable to use in this environment.

1.2.1 Point to Point

With this model communications is carried out on a one to one basis. Sockets are an example of a point to point communications paradigm. To use this type of mechanism the two parties to the communications must know before communicating who they will be communicating with, where the communication will take place, and when the communication will happen. This a priori knowledge may not always be available in a mobile environment.

1.2.2 Group Communication

Group communication allows a set of members of a group to be addressed as a single entity. This type of model normally involves some form of multicast to direct the

messages efficiently to all members of the group. The model can impose a number of ordering semantics to the delivery of messages to members of a group. Types of ordering include, total, causal and none. Total ordering imposes a sequencing on message delivery, which is identical for all members of a group, In causal ordering, messages may not be fully ordered but all message will be delivered such that the causal orderer of each message will be the same throughout the group. Finally groups can be created were ordering is not imposed. This type of communication paradigm is very useful for fault tolerant maintenance of distributed databases. ISIS [5] is an example of a system that implements this paradigm. In many respects a group which is made up of members is very similar in nature to a worm, which itself is made up of segments.

1.2.3 Events

Events are a mechanism by which a party can communicate with other interested parties, informing them, that some form of state change has occurred. Events are both asynchronous and anonymous, the source has no knowledge of its subscribers and a subscriber has no knowledge of its sources. Events can, and often do, have a one to many relationship and this type of paradigm allows us to de-couple the sender from the receiver. Events are also asynchronous, the event is generated and the party that generated the event can immediately resume its work. Events only require a priori knowledge of when to communicate. They do not need to know who they are communicating with or where communications should be addressed. This is known as a loosely coupled communications model.

1.3 Fault Tolerance

A system can be made fault tolerant by adding hardware or software redundancy. If a critical database must continue functioning in the face of failure, one of the easiest ways to ensure this is to replicate both the hardware and software over a number of different locations. If one server fails, another will takes its place automatically

and system users will not be aware of the failure. This simple solutions works well where the data is relatively static, for instance an on-line parts database whose data is reasonably fixed and where updates can be scheduled during quite usage periods. If on the other hand the data is dynamic then keeping all the replicas updated becomes a difficult problem.

Replication can also be achieved using software. The simplest solution is to use hot shadows. A hot shadow is a duplicate of a process that responds to messages but does not itself, generate any messages. This will of course tie up resources, constantly maintaining the shadows as well as the main processes. A difficulty with hot shadows occurs when messages have causal dependencies and no message ordering semantics have been imposed. This situation may lead to an inconsistent state between a segment and its shadow. Group communication §1.2.2 as we have seen can be used to maintain consistent replicas, a system such as ISIS will ensure correct causal ordering of all messages to members of a group.

One of the features of a worm is its ability to recover from segment failure. The paradigm used to implement fault tolerance in a worm will depend on the requirements of users, and the type of computations they run over a worm. Using a worm to host a number of independent computations, where time is not critical, requires only that dead segments be recreated, and calculations be restarted again. At the other extreme, a time critical and complex calculation with dependencies across all segments, may well require some form of group communications to implement the desirable level of fault tolerance. Because this dissertation is mainly concerned with using the event paradigm in a mobile environment, the fault tolerant requirements of the worm have been kept to a minimum.

1.4 Migration

Fundamentally a worm requires migration. First it must be able to replicate its required compliment of segments, and then each segment must be migrated to a new host. Migration as will be seen in §1.5 is also a requirement if the system is to avail of

load balancing facilities. As a result of using Java in this implementation, migration takes advantage of the built in serialization facilities available in Java.

1.5 Load Balancing

One of the benefits of using a worm is to take advantage of under utilized workstations throughout the system. To gain the most benefit from the system the distributed computation should take advantage of as much processing power as is available across the network. To do this effectively requires balancing the computational load across all workstations available on the network. There are two separate parts to load balancing, first the loads on each available workstation must be available. From this a decision must be made as to which is the least loaded workstation. When this workstation is known then a process residing on a heavily loaded workstation can be migrated to the least loaded workstation. Servers hosting the worm segments will use load balancing, allowing the worm to take full advantage of the processor resources across the network.

1.6 Objectives

In the light of the environment portrayed in this introduction, the dissertation takes a new look at the possibilities now available to distribute computations over a number of idle workstations. In particular it looks at how we may exploit the worm paradigm in this new and dynamic environment. To exploit this environment the worm will provide four major services to its clients:

1. Transparent migration.
2. Location independent communications.
3. Segment fault tolerance.
4. Computational parallelism.

Delivering these services to its clients, should be achieved using the following constraints:

- The worm should be fully distributed.
- The code to implement the worm should have a small footprint.
- The worm should be easy to use.
- The worm should use an asynchronous and anonymous communications model.
- The running worm should require the minimum amount of compute and communications resources.
- The worm should function in a heterogeneous environment.

Working within the bounds of the above requirements and constraints, the dissertation details the design and implementation of a number of services required to create a worm capable of being hosted in a mobile environment. The five major services designed and implemented are:

- A segment hosting service to allow segments to be hosted on a machine.
- A migration service to allow worm segments to be moved across machine boundaries in a transparent manner.
- An event service to allow worm segments to communicate in the presence of a dynamic environment.
- A worm that can continue its computations in the face of segment failure.
- A load balancing service to allow the worm to maximize its use of processor resources in the virtual network.

Having designed and implemented the above services, a number of applications using the worm paradigm were designed and implemented with a view to confirming the premise that events would indeed allow a worm to function well in a mobile environment.

1.7 Conventions and Notations

This document deals with the development of a library of objects that enable the user to implement a worm in a transparent manner. Wherever possible the conventions used to display code use, `typewriter type` algorithm descriptions use *italics*.

The word user in the context of this dissertation normally refers to a person who is using the code developed for this dissertation, to implement a distributed program using the worm paradigm.

1.8 The Working Environment

The Worm should capable of deployment in a heterogeneous computing environment. It will be developed in Java, and should therefor work under any operating system that runs a Java Virtual Machine.

The event model will use the TCP/IP protocol suite as its underlying network protocol. The model will take advantage of IP Multicasting to overcome the location dependency problems of a mobile environment.

There will be no requirement for a shared file system such as NFS, as the system will be capable of transporting all its code and data internally using its own protocols running over TCP/IP.

Initially for test purposes, each host server will have a graphical front end that will allow a user to force a segment to migrate. It will also allow a user to kill segments residing on a host server. This in turn requires that under Unix an X server is present. In a production version of the software there should be no requirement for each server to have a graphical front end, and one can envisage writing a protocol that allows a user to do all maintenance from any host server within the virtual network.

A virtual network in the system is defined as the set of all host that are running servers capable of hosting worm segments. In 1.1 an example may be seen of a network on which three hosts are running servers capable of hosting worm segments. The virtual network therefor comprises these three hosts. A virtual network can span

many individual networks.

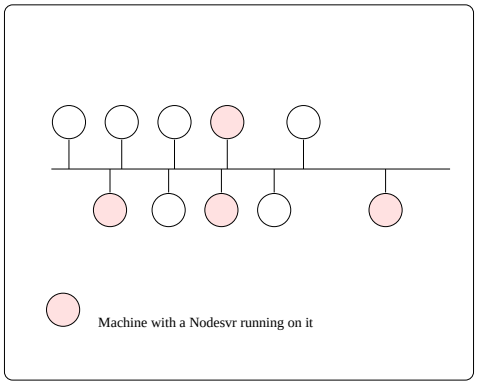


Figure 1.1: The worms working environment

1.9 Document road-map

Chapter 1 Introduction

This chapter describes general details about distributed systems and about the worm systems in particular. Motivation for doing this project is also covered along with a brief introduction to communication and migration requirements of the system.

Chapter 2 Distributed Systems Programming

This chapter takes a brief look at some of the general development problems encountered when building distributed systems. It looks at some of these problems as they were encountered in the context of building a worm.

Chapter 3 Technologies useful to a worm

This chapter surveys a number of existing technologies that are available today, and why they may or may not be suitable to use in the development of a distributed worm.

Chapter 4 Object Movement

This chapter takes a look at the problems that may be encountered implementing Object movement and process migration in a heterogeneous environment. It goes on to survey a number of general technologies. Because the worm is developed in Java the chapter finishes with a detailed analysis of Object serialization and class loaders in Java.

Chapter 5 Analysis and design

This chapter builds a set of design requirements for the distributed worm.

Chapter 6 Implementation

This chapter presents the detail of how the design requirements developed in chapter 5 are implemented using the Java programming language. The chapter takes a brief look at some of the more important classes in the system and their inter-relationships.

Chapter 7

This chapter takes a look back over the design with the benefit of hind sight, It evaluates effectiveness of the worm using an events paradigm in solving real world problems using the applications developed.

Chapter 2

Distributed Systems Programming

In a perfect world, distributed programming should present no more difficulties than programming a centralized system. We do not have a perfect world, and must contend with the added complexities a distributed system exposes. This chapter examines some of the complexities introduced when a system is distributed. These new variables, introduced by distribution, are usually external to the system, but normally play an important role in the system design. The worm is of course a distributed system. Designing and implementing the worm requires understanding the general problems, and solutions to those problems, put forward in this chapter.

2.1 Naming

Naming is one of the most important attributes of any system, distributed or centralized. To gain access to, and use of, a resource it must be addressable. In a small network of workstations such as a Local Area Network, naming is often arbitrary, a printer may be assigned a name related to its location in a building, as in the name wr13hall, this name represents a printer residing in the hall of 13 Westland Row. Other naming schemes often use a serial number chosen as new resources are added to the network, yet more are chosen by naming them after famous people or loved ones. This type of naming works well in small centrally controlled environments, but as the environment is scaled up to meet the requirements of a wide area network or

Internet, problems with name clashes occur with ever greater frequency. To address the scaling problem, names are often divided into self administered domains, and these domains are linked together to form a hierarchic tree structure as shown in figure 2.1. A good example of this type of structure is found in the name service used within the Internet, The Domain Name Service or DNS for short.

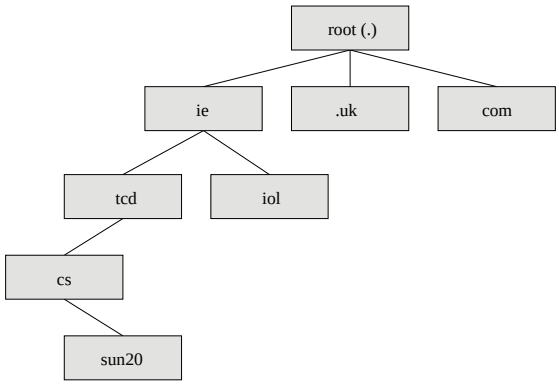


Figure 2.1: The hierarchic nature of DNS

2.2 Consistency

Consistency is a good defining property of all systems, not just distributed ones. A consistent model builds a familiar environment in which the application programmer, and users alike,can work. It allows the user to build on an ever increasing knowledge base, and when presented with new extensions to the model, to use them with confidence and understanding. An example of where consistency may be used to good effect, is the development of a good error handling model. Handling errors can be a difficult task, and in a distributed system this task becomes critical to the correct behavior of the system. The user of a system should be presented with a consistent model for error handling, they should not find themselves in the position that one method returns a -1 to indicate an error but another similar function throws an exception in response to an error.

2.3 Transparency

Transparency is used to hide the internal workings of a system from its users. CORBA is an example of how transparency in method invocation of distributed objects appears to the user. Calling a remote object's method using CORBA, after having bound to it, appears to the user identical to calling a local object, even though the remote object may live on a machine thousands of miles away. This type of transparency makes using a distributed system appear to the application programmer, identical to using a local system.

This transparency does have some drawbacks[41], invoking a method on a remote object does not have the same call semantics as invoking a method on a local object. When invoking a local object the semantics of the call are, exactly once, whereas the semantics of calling a remote object may be, at most once, at least once, or some other combination imposed by the system designer. These differences may be important to the application programmer in the face of failure. Invocation transparency should be used to ease the transition of current systems to a distributed paradigm, but not to hide the fact that a system is distributed.

Location transparency enables resources to be accessed without the user requiring knowledge of their location. In many respects this type of transparency is tightly bound to the naming service used within the system. The worm project uses the event paradigm, in tandem with its naming service, to isolate the addressing and use of resources, from their locations. This type of transparency takes on added importance in mobile environments, where resources may migrate many times over the period of a single computation.

Migration transparency is another desirable property. It allows a processes to be migrated to a new host without the process requiring any knowledge of, or having to take any special action, as a result of the migration. In the case of the worm full migration transparency would not been possible without moving the code responsible for migration into the Java Virtual Machine. Time has not allowed this route to be explored so a compromise solution has been imposed. The worm requires that a

worker has a well defined structure, imposed by forcing the worker to implement an interface. This solution is very similar to the Java Applets design.

2.4 Time and State

Time in a computer system, as in the world, is the dimension that measures change, if a systems state never changed, then time has no meaning to that system. Change is observed in a system by parts of that system changing state.

As a simple example, an object T has one variable *count*, when T is created, *count* is set to zero. If throughout the life of T the variable *count* is never allowed to change, that is, *count* is immutable, then T has not changed state and time for T in this system is not relevant.

On the other hand if *count* is allowed to change and is incremented every n seconds then the state of T changes every n seconds and time is used to measure when a state change occurs.

In a centralized system the internal processor clock marks time, and all objects within the system have a consistent view of time and therefor state. In a distributed system, clocks on separate computers within the system may not be synchronized or the resolution of their clocks may be different. Also the problem of unpredictable communication latency makes any form of clock synchronization difficult. Without a synchronized view of time, a consistent view of the systems state at any point in time is not possible. Problems of synchronization have been solved using mechanisms such as logical clocks [26] or vector timestamps, for a good general discussion of how distributed systems overcome these synchronization problems see [35] Chapter 4.
The worm has only five views of a segments state;

1. The segment is waiting for work.
2. The segment is working on a problem.
3. The segment completed its work.
4. The segment has finished all work and terminated.

5. The segment has suffered premature death.

The worm has a very limited view of a segments state. If a segment dies, all knowledge of its current state dies with it. The worm when it recreates the dead segment puts the new segment into a waiting for work state, even if its ancestor had nearly finished a large computation, this knowledge will be lost. To enhance the worm so that the segments internal work could be saved at key stages in the computation requires the worm to have state synchronized across all segments. With this synchronization in place a checkpoint mechanism could then be used to take a consistent cut of the current state of all segments making up a worm. The checkpoint data could then be saved to persistent storage, to be recovered at a later stage if required. This checkpoint mechanism appears simple, but distributed check-pointing and obtaining a consistent cut are difficult problems [25].

Having a global view of time in the system is still not enough to guarantee a consistent checkpoint of the worm. State changes in one segment that may effect the state of other segments are communicated using events. Events designed in this model are asynchronous, no protocol has been implemented to guarantee the order in which events arrive at any segment. This means in the current model, state changes do not propagate to all segments in a consistent manner. To enforce this type of consistency requires events be delivered in a causally ordered manner [4].

2.5 Lack of global Knowledge

Lack of global knowledge is a problem faced by all distributed systems and can have a major bearing on the design of a system. In a tightly coupled system running on a single host, the state of all parts of the system are known at all times. This is not the case when a system is distributed. the classic example of this is known as the two generals problem [45]. In essence when two computers communicate, it is not possible to be absolutely certain that the final message was received.

As pointed out in §1.8, each worm host on the virtual network has no knowledge of the existence of other worm hosts. By designing the system so no state is held about

the topology of the virtual network, that topology may change dynamically with little or no impact on the running of individual servers. Problems arise in the design when a node becomes overloaded and wishes to migrate a segment. Under these conditions a way must be found to locate other servers in the virtual network.

Servers use IP Multicasts to locate a suitable target for migration. The current node multicasts across the virtual network requesting all nodes return their current load status. This may appear to solve the problem, but multicasts are unreliable and the current server has no idea how many other server's are available on the Virtual Network. If three replies are returned as a result of a load request the server cannot be sure if this means there are three other nodes on the Virtual Network or that only three nodes actually received the multicast. To overcome this problem of partial knowledge we can do one of three things:

1. State can be maintained in each worm host describing the makeup of the network, giving each worm host full knowledge of network topology, this solution impacts on the dynamic nature of the network, each worm host must keep track of when other servers join, leave or fail. Keeping track of joiners and leavers is a simple matter, tracking failed servers requires an added layer of complexity.

2. Implement a reliable multicast protocol, which again adds greatly to the complexity and reduces the speed of the system.

3. Use Group membership to maintain a consistent view of the virtual network. Group paradigms a normally very heavy weight.

Another example of lack of global knowledge in the worm is detection of segment death. The user when creating a worm specifies a pulse rate. This pulse rate is used by segments to generate heartbeats. At each heartbeat the segment raises an event that another segment, known as its active monitor will receive. If the active monitor does not receive a predetermined number of consecutive heartbeats within the alloted time then the segment is deemed to have died. Has the segment really died or has the machine on which its running just become very slow for a short time, or has the

network partitioned. The active monitor cannot know which of these outcomes are true. It will only know if the second or third outcome is true when the supposedly dead segment comes by to life along side its newly created counterpart.

The above examples of lack of global knowledge are by no means complete but they show that awareness of lack of global knowledge must be designed into a distributed system from its inception. Algorithms within distributed systems must be designed to handle problems that arise from this lack of global knowledge.

2.6 Fault Tolerance

It is a fact of life that computer systems fail. Failure can occur in many parts of the system, and for many reasons. A computer may fail due to its power supply overloading. A network may fail due to wiring problems. Software may fail due to a bug resulting in exceptional conditions not being catered for when a user submits garbage from a form to a processing program. It is not possible to have a priori knowledge of failure. What happens to a system after a failure is also often unpredictable. The best one can hope for is that the part that fails will stop. A very difficult failure to account for is Byzantine failure. This type of failure will cause a system to continue functioning but the output from the system will be erroneous.

To cope with failure, distributed system are built to withstand a certain degree of failure, that is they are fault tolerant. Achieving a fully fault tolerant distributed system is very difficult. The level of fault tolerance required is usually dependent on the system and user requirements. In general the more state held by each part of the distributed system, or the rate at which state changes, the more difficult the job of making the system fault tolerant becomes.

In the case of the worm, and in particular the matrix multiply application, fault tolerance in the form of the re-creation of segments after death, would appear a trivial task. As has been shown, the problem of partial knowledge adds greatly to the complexity by injecting an element of uncertainty into when we can declare a process or segment dead. We must cater for the exceptional case of the dead process

mysteriously re-appearing. Good examples of algorithms to handle this situation can be found in the ISIS system[4] when a member of a group fails.

2.7 Concurrency across machines

Controlling concurrency on a single machine can be difficult. There are no fixed rules that can be applied to a concurrent system at the design stage that will guarantee the code is free from either deadlocks or race conditions.

A deadlock in a concurrent system occurs when a thread A holds a lock on a resource X, but requires access to resource Y to complete its computation, unfortunately resource Y is all-ready held by thread B, which in turn requires resource X to complete its computation. If a resource dependency graph were constructed for this situation, then there would be a cycle in the graph. In this circumstance the only way to proceed is to make one of the process's relinquish its locks and restart its computation at a later time. Many deadlock situations can be avoided by using a strict locking protocol such as strict two phase locking from the field of transaction processing.

Race conditions occur when two threads require access to the same variable. If a thread A wishes to update a variable x by adding 100 to it, and thread B also wishes to update variable x by adding 50 to it, then if x is Initially 200 we would expect the final value of x to by 350. This may not be the outcome, if both read the variable at the same time, so they both get out the value 200, now thread A writes back 200, after this B writes back the value 150 and this is the final value of in x at the end of the computation.

To avoid race conditions, all shared data should be protected with some form of synchronization. Adding synchronization reduces the liveness of a system and increases the possibility of deadlocks.

In a distributed system these concurrency problems can occur across a number of machines within a network. Avoiding distributed deadlock and race conditions involves developing strict locking protocols.

Liveness can be a problem when communicating across a slow network. In early incarnations of the worm all threads in the system ran at the same priority, this situation often lead to long delays when a server tried to move a segment or event from itself to another server. One of the reasons for these delays happened when either the receiving or sending thread was preempted leading to timeouts at the other end of the network connection. To overcome this problem in later iterations of the worm, a form of priority scheduling was implemented. All threads that handle network traffic are given a high priority to ensure that when a segment or event is moved across the network the threads on both machines are not preempted.

2.8 Debugging and testing

Good debugging tools are no substitute for a good design. A debugger no matter how powerful or easy to use, will not in itself expose design flaws. Taking cognizance of the above, when developing large or complex systems, debuggers make the task of tracking down errors a relatively painless operation. In developing the worm, the lack of debugging tools for distributed systems has become apparent. Debuggers for Java are available but they can only be used to debug code running on one machine. In distributed systems many problems occur as a result of subtle timing errors due to incorrect protocol specification. The only tools available to the developer at this point appear to be outputting critical data to log files and then using grep and awk to analyze the log outputs from a run.

Debugging has often been referred to as an art not a science. This is not true, to debug a system effectively using only log files, or a state of the art visual debugging tool requires the person debugging the system to have a broad understanding of the system at an abstract level. In essence a reasonable debugger in the hands of an expert is far more effective that a state of the art debugger in the hands of a novice. For a discussion on levels of expertise see [14].

Testing has also proved difficult, especially when using a busy network. Busy networks do not allow you to measure a lower bound for system performance. Busy

networks may also unpredictable and any quantitative analysis is rarely consistent from run to run.

This chapter has looked at a number of the problems that a distributed system designer may encounter when designing a new system. It has put some of those problems into the context of the design and implementation of the worm. Many of the properties discussed, such as naming, consistency, transparency and fault tolerance are applicable to the design of the worm. This general discussion along with the next chapter's survey of available technologies will put in place the ground work for the general design specifications of the worm.

Chapter 3

Technologies useful to a worm

This chapter looks at a number of existing technologies that assist in the implementation of distributed systems. The purpose of this survey is to investigate if the worm can avail of the services provided by any of the current systems, and as a result, cut development time.
The worm should provide three major services to its clients:

1. Transparent migration.
2. Segment fault tolerance.
3. Computational parallelism.

Delivering these services to its clients, should be achieved using the following constraints:

1. The worm should be fully distributed.
2. The code to implement the worm should have a small footprint.
3. The worm should be easy to use.
4. The worm should use an asynchronous and anonymous communications model.
5. The running worm should require the minimum amount of compute and communications resources.

6. The worm should function in a heterogeneous environment.

Developing a worm that can provide all the services required, while meeting the above constraints may not be feasible within the time-frame of this project. However it will provide the necessary framework on which to structure our survey of current systems. The survey takes a particular interest in migration and event services because both of these technologies are fundamental to the development of the mobile worm.

3.1 The Worm Paradigm

The first idea for a computer worm comes from a novelist J. Brunner [7]. In his book The Shockwave Rider, Brunner puts forward the concept of a tapeworm. This worm runs loose throughout a network of computers. The worm can spread segments of itself across many computers, if one segment dies the worm recreates it. This omnipotent worm can take over a network, commanding all the networks resources for its own ends. Trying to kill the worm on individual host is usually futile. Although this concept is unsettling, it does have merit it terms of a mechanism that may be used to distribute work across a network of workstations, many of which are idle for long periods of time.

3.1.1 Shock and Hupp, Early experiences

Shock and Hupp [42] saw the positive benefits of the worm concept described in Brunner's work. They set to work to create a worm that, rather than being malicious, could be used as the mechanism to distribute useful computations. Their concept started off as a blob that could move from workstation to workstation, eventually finding an idle one. Here it would replicate and leave a copy to expedite a computation while it continued its search for more resources. Following on from this they built a series of worms with ever increasing functionally.

The first worm they built, know as an existential worm had as its sole purpose in life, existence. It was a test of the concept and the ability of a program to duplicate

and migrate over a network of machines. The Billboard worm was used to carry a small graphic, the cartoon of the day, around a group of workstations. An alarm clock worm was also written which could be used to display a message to a user, who had been previously setup the worm to display the message at a set time. The final test was a much more ambitious worm, The Multi Machine animation worm worked as a master slave system, the master was used to farm out animation work to slaves, receive back the completed work, and collate and show the final animation.
In the worms that were built, a number of issues were hi-lighted:

- They proved the concept worked.
- The communications paradigm was an issue. They eventually settled on a form of brute force multicast where each segment sends to all other segments. The cost of such a system is $n(n-1)$ where n is the number of segments in the worm.
- Controlling the spread of a worm is important. They had found that where the creation of segments was unbounded the worm could grow exponentially.

In conclusion there work has shown that the worm paradigm works, but attention to detail in its control mechanism is essential. Communication remained an issue, but this issue can be resolved using a newer paradigm such as events or tuple space, both discussed later, that had not been available to them at the time of their research. The other point that should be noted from this work is that the worms were only designed to work in a homogeneous environment, a set of Alto computers linked together on an Ethernet network.

3.1.2 The Internet worm

On November 2nd 1988 Richard Morris, a post-grad at Cornell University released a worm onto the Internet. This worm was based entirely on the concept portrayed in Brunner's book, and was designed with the sole purpose of existence and replication, details of the attack and the design of the worm are documented by Donn Seeley [40].

The worm was designed to defend itself against attack, to probe unwilling host with a view to migration and when migrated to move on to its next victim. To do this the worm, exploited loopholes in the security of both VAX and Sun-OS operating system. The worm was divided into two parts, the first a simple bootstrap program, moved onto a new host and established a process, from there it could down-load the second and bigger part, which would then go about the business of duplication and migration.

The major problem with the Internet worm was, as Shock and Hupp had found earlier in their experiments, the worm, without some form of replication control, would reproduce without bound. This multitude of worm all replicating exponentially eventually brought many host on the Internet to a stop. It took a number of days to remove the worm from all hosts it had infected. Many machines had to brought down and re-booted.

This episode has done untold damage to the public view of the worm paradigm, and little more research has been conducted into using the worm paradigm as a reasonable means of distributing computations. Taking aside its public persona the worm as a distributed tool still has much to offer, to this end exploring its potential is a reasonable research aim.

3.1.3 Setanta Mathews, Reliability using a Worm

As a final year project for his B.A. (Mod) Mathews designed and implemented a worm [34] that used hot shadows for each of the worms segments to enable a reasonable degree of fault tolerance. Mathews also used the event paradigm to build the communications info-structure for his worm. He designed a number of experiments, including matrix multiply and Conway's game of life to test the worm.

The design allowed for the presence of a shared file system, and the event service was implemented using the Orbix event service, a commercial CORBA event service implementation by IONA. There was no serialization built into the system, therefor if the hot shadow failed the worm could not continue. Mathews points out this weakness in his section on future work.

The Orbix event service had at the time no support for typed events, so Mathews was forced to build a pseudo typed event over the untyped events provided by Orbix. Mathews notes in his conclusions that the Event service was centralized, creating a single point of failure if the server goes down, or is partitioned. He also noted that the event service was slow and at times unreliable.

3.2 Migration

Migration is the ability to move a running process from one host to another. A worm by its nature requires some form of migration to exist and replicate. Migrating a process come in numerous levels of sophistication. In its simplest form, the migrated process may be moved to a new host but will make callbacks to its original host to maintain resources that could not be migrated with it, these may include files opened on the original host's file system. The holy grail for migration is the ability to migrate a running process transparently, leaving no residual dependencies on the original host. There are a number of good reasons why migration is a useful tool:

- Balance the load across a cluster of workstations.
- Move a process to the source of its data.
- Improve program resilience by insuring the failure of a host does not stop a process.
- Move a process to a host that has specialized resources that the process can utilize.

Migration as the above list shows has many benefits, those benefits come at a high price in terms of the implementation cost of achieving a transparent migration service. Some of the issues that must be dealt with are:

- Naming. Normally a running process on a workstation is identified by its process ID. When a process migrates then this ID is not valid on the new machine, a new ID which is relevant to the new host must be assigned. If this process

communicates with other processes on the network then some form of naming independent of location or Process ID is required. Steve Benford and Ok-Ki Lee [3] discuss the difficulties of naming in distributed systems, and put forward a model for overcoming these difficulties.

- Migration Latency. A process will take a finite amount of time to migrate, within this period it may receive messages.

- Communication. Communication is as we have seen , dependent on some form of general naming to deliver messages. Another difficulty is the requirement to store and forward messages due to migration latency.

- Security. Can you trust a process to behave in a reasonable and safe manner when it cohabits with other processes on the same workstation. If not how can the migrated process be isolated so even if its behavior is malevolent, this will not effect the running of other process on the host.

The next three subsections give a very basic introduction to a number of currently available migration facilities.

3.2.1 Classic kernel supported migration

The V System [10] is an operating system kernel designed to facilitate distributed systems. It has migration services built into the kernel. Theimer et al [33] discuss the kernel based migration facilities. Naming is implemented using a logical naming scheme, this allows a process to maintain the same logical name regardless of where it migrates to.

The migration service overcomes the problem of latency by using a technique called, Precopying. When a process is to be migrated the kernel will precopy a bootstrap process onto the new host. This process will Initially have a different logical name to the original process to allow both processes to use standard IPC calls to move the bulk of the process state. This ID will be changed later to the ID of the original process once the original process has been removed. During migration

the bootstrap process has enough state to accept and hold communications. This allows both processes to accept communications at the same time. The old process can forward any messages it has to the new process at the end of the migration. Duplicate messages will be checked for, and ignored by the new process.

DEMOS/MP [32] is an operating system that also has a built in kernel migration service. It does not use a logical naming system but instead leaves a forwarding address with the original host. This type of arrangement suits a system were migration only happens rarely, it does not scale well.

The Chorus system is a micro kernel architecture and some work has been done on migration in such an environment. The paper by O'Connor [38] covers some of the details of migration under a micro kernel. Many of the problems are common to both standard and micro kernel architectures. The implementation uses a store and forward mechanism to overcome migration latency. The implementation also takes advantage of the paging architecture to supply memory pages still residing on the original host, using an, on a demand only basis.

3.2.2 Migration outside the kernel

The Condor [31] libraries allow processes in Unix to be migrated without building the migration facilities into the kernel. This means the system can be readily ported to many variants of Unix without requiring access to, or modification of the kernel.

Condor uses the concept of a system call re-director to maintain links with its original host in a transparent manner. It uses check-pointing and the use of core dumps to stop migrate and restart process as transparently as possible.

3.2.3 Java migration

Using the build in serialization package, Java objects can be migrated with little difficulty. Two example Agent system that both use these techniques are Aglets [27] from IBM and Voyager [13] from ObjectSpace.

Migrating an object is not the same as migrating a process. In process migra-

tion state on both data, and threads of execution are moved intact. When migrating objects in Java only the object state is migrated, the thread and execution stack and therefor the state of all local variables are not migrated.

There is at present some work being done to support the migration of threads in Java. Two approaches are being investigated. The first is to modify the Java Virtual Machine to save and restore the state of any of its running threads. Taking this approach means the user having to install a non standard Java Virtual Machine on their workstation. An example of this approach can be found in the Nomads project.

The other approach, put forward by Funfroken [48], is to instrument code using a Java precompiler to insert code to save the object and stack state at critical points. This meta data can be used to re-assemble the execution state of a thread after migration. The advantage of a system like this is, its transparent to the user and has no requirement for a special Java Virtual Machine.

This section has covered types of migration implementations, the classic kernel migration mechanisms are of little use to the current project. Having said that, they are useful as a source of algorithms and solutions to common problems that are encountered in any form of general migration implementations. The work being done at present on Java migration is only in the very early stages, and would not be useful in this project. None of the existing technologies surveyed would prove useful to the implementation of the worm, however much detailed knowledge has been gained from the common experiences of migration in general. This detail will be useful in the design of the worms migration service.

3.3 Load Balancing

An efficient worm should make optimal use of all resources available to it. One of the major resources in a system is processor time. A worm whose segments all reside on a single workstation, while many other workstations with lower loads are available to it is not using its computing resource optimally. To work efficiently the worm should

distribute its segments across the least loaded workstations available. In [18] Guy Bernard et al take a close look at a number of load balancing algorithms available and there efficiency. The simplicity and efficiency of an algorithm such as LEAST would suit the load balancing requirements of the worm.

3.4 Communications

Communications is a fundamental requirement for a worm. Communications normally involves three properties, Who, Where and When. Who or what do we wish to contact, where can they be contacted and when can this contact be performed. Most system require that all three parameters be known prior to communication. Multicast eases the situation by not requiring who, Events go further by not requiring a prior knowledge of who or where, and tuple spaces[8] allow communication without knowing a priori who, where or when. This section surveys a number of technologies that may be used to provide an info-structure for the worms communications requirements. Starting with the lowest level, TCP/IP is briefly reviewed. CORBA because of its current pivotal position in distributed computing is examined. Events are central to the theme of this dissertation, therefor a number of current event models are reviewed. Finally an old paradigm, tuple space, which was the communication paradigm used by Linda, is reviewed in the light of JavaSoft developing their own tuple space model, JavaSpaces. This model although not available as of this survey should be released in the last quarter of 1999. The tuple space model appears to be an interesting alternative to events.

3.4.1 TCP/IP

The TCP/IP [11]protocol suite has been with us for a long time. It is the core protocol of the Internet, and has made major inroads into the LAN arena. The main interest from the point of view of designing a worm, are the error semantics associated with each of the general protocols.

IP the fundamental protocol, is an unreliable, best effort connectionless packet

service, it therefor makes no guarantees of delivery, ordering or correctness of packet data when and if it arrives.

UDP is also a datagram service, which imposes the same delivery semantics as IP but UDP adds the ability to deliver datagrams to multiple known destinations on a host by using the concept of a port as its endpoint.

TCP is a reliable stream protocol, it guarantees that the data sent will arrive, and when it arrives it will be in order and the packets contents will be correct.

3.4.2 Java RMI

Java Remote Method Invocation [1], RMI for short, is based on the classic RPC model [6]. RMI allows a Java object on one host to make method calls on Java objects resident on other hosts. This type of setup normally uses a proxy for the object being called, which is resident on the local machine. This proxy serializes all the methods parameters using Java serialization, this data is then transported across the network to a skeleton method on the destination host. The skeleton makes the actual call to the object on the destination host, gets back any parameters, serializes them and transports them back across the network to the calling hosts proxy. The proxy in turn returns them to the calling object, see figure 3.1.

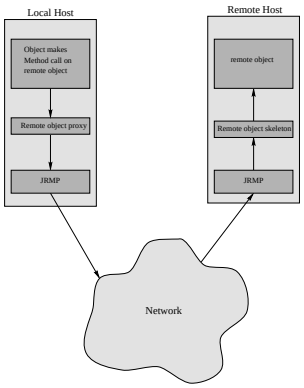


Figure 3.1: The Java RMI architecture

Java RMI uses its own protocol, Java Remote Method Protocol (JRMP), built over TCP, to transport method information, and the serialized parameters across

the network. The remote objects must be on the destination host and registered in the Java RMI registry before they may be called. RMI functions only in a Java environment.

3.4.3 CORBA Remote Method Invocation

The CORBA specification [17] is designed to provide a general language and architecture neutral solution to making remote method calls between objects written in different languages and resident on separate architectures. In essence in provides the same services as Java RMI, but can inter-operate across many platforms and languages.

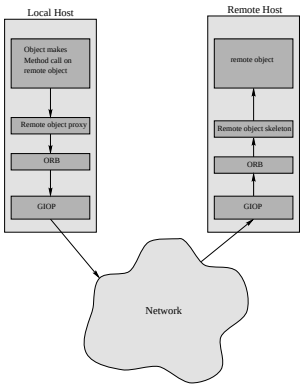


Figure 3.2: The CORBA RMI architecture

The core of the model is an Object Request Broker or ORB. The ORB acts as a an intermediary between the calling and called object methods. It functions in a language neutral manner by requiring all remote interfaces be declared using an Interface Definition Language. This design allows CORBA to be used as a general purpose middle-ware product in distributed system. It takes care of much of the house keeping code required in a heterogeneous environment. Vinoski [47] gives a good overview of the CORBA specification, use and implementation in heterogeneous environments. CORBA is in essence a client server middle-ware product, it was not designed to be deployed in an environment where the remote objects were mobile because once you bind to a remote object, part of what is returned to the proxy is the host on which the remote object is resident. Even if the remote object could

migrate, which it cant, the proxy would still only have the original host on which the remote object resided.

3.4.4 Events

Bacon et al [24] describe events as "an autonomous asynchronous occurrence". Events are normally used by an entity to inform other entities that some type of state change has occurred. The concept can be seen in industrial settings where a pressure sensor in a process line triggers an event if it goes above a preset threshold, that event will eventually lead to a valve further up the line being adjusted accordingly.

In computer design an interrupt is designed to handle asynchronous events generated by changes in the hardware or software state. Events like interrupts are asynchronous.

Software events in general allow the sources and subscribers to an event to function without any knowledge of each other. This means in the who, when and where scenario, events only require the when constraint. To receive an event you must have subscribed to that event prior to the event being raised. The paradigm also allows for a number of, subscriber to source relationships:

- one to one.
- one to many.
- many to many.
- many to one.

There are two general event implementation strategies, push and pull, both are concerned with the direction of flow of the event as it travels through the system. In the push model events are pushed from the source through the system toward the subscribers. In this model shown in figure 3.3 the source is the active component in the system, and the subscriber passively waits until events arrive.

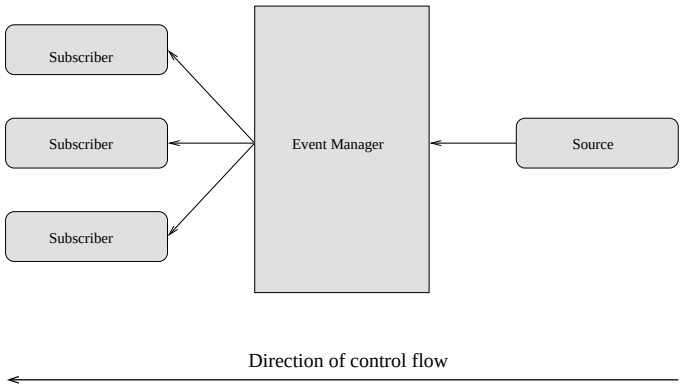


Figure 3.3: The push event architecture

In the pull model events generated by the source are often queued at the source and it becomes the responsibility of the subscribers to poll the source and pull down from the source any events queued for them. Figure 3.4 shows this arrangement.

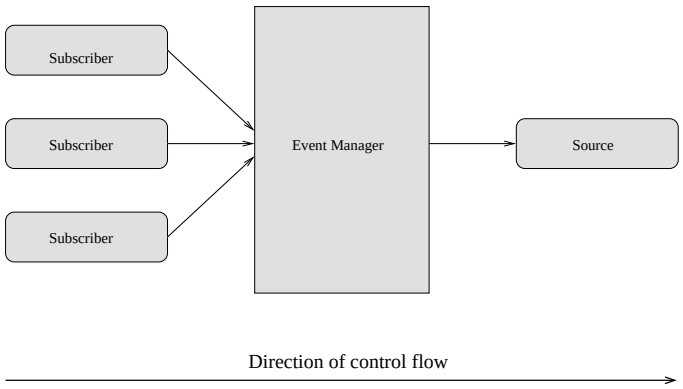


Figure 3.4: The pull event architecture

The Java event model

Java implements a rudimentary push event architecture. Events are normally used to propagate actions that occur in the GUI [9]. The system is much closer to an interrupt mechanism, where for each type of GUI event, such as a user pressing a button, the application can install a callback object that the GUI manager will call in response to the required event happening. The system only implements a one to one relationship. Java does not support any concept of distributed events at present.

The CORBA event bus

The OMG has defined a specification for the implementation of distributed events [21]. As in the CORBA model this event model has been designed to operate in a heterogeneous environment. CORBA uses the concept of an event bus. The event bus abstracts the subscribers from the sources. All interfaces to the event channel are created using an IDL. The event channel supports both untyped and typed events, but at present not all orb implementations support typed events. Both the push and pull models and variations of both can be implemented on a single event channel at the request of either the subscribers or sources.

The event channel is centralized on a server and from the worms point of view leads to a single point of failure. Also because the worm is using Java as its means of coping with heterogeneous environments, and has no requirement to communicate with objects written in a language other than Java, the CORBA event service would prove an expensive option with little gain.

The ECO model

The ECO model [44] is an event framework designed for object oriented systems. It is made up of three parts, events, constraints and objects. the model builds on the general events model adding to the model, the concept of constraints. ECO defines three types of constraints, Notify, pre and post. Notify constraints are supplied by a subscriber informing a source that this subscriber wishes to receive events of a certain type only if the supplied constrain is matched. Evaluating notify constraints is done at the source. A situation may arise where an object monitoring room temperature only wishing to be informed of a change temperature event, if the temperature variance is greater that 1 degree. If the resolution of the monitoring device is say .1 of a degree, small variances will not propagated, so saving on event generation, and as a result network bandwidth[19].

Two other forms of constraint may be implemented. The preconstraint is evaluated just prior to the event being delivered to a subscribers callback method.

Preconstraints could be used to enforce an ordered event queue without the developer having to modify the event service code. The post constraint is evaluated just after the call to a subscribers callback method.

The Cambridge model

The Cambridge model [24] builds further on the above models and allows the concept of composite events. A composite event is where an event is only generated if a number of other events evaluate to true. The system can be implemented using a form of finite state machine, each state transformation occurring as a result of a certain event type being generated. Event ordering in this system is important due to the nature of state transitions in an FSA.

Tuple Space

The Linda system [8] defines the concept of tuple space. A tuple can be defined as a set of types, not unlike the signature of a method. A tuple may be defined as {String,float,int}, an instance of this tuple, {Test-tup,143.92,001} could then be entered into tuple space. If another entity is interested in receiving this tuple it must present a template in the form of the original tuple {String,float,int} to the tuple space. The tuple space on receiving this template will fill in the data and return a tuple, {Test-tup,143.92,001} to the calling entity. Tuple space is persistent, therefor unlike the event model where a subscriber only receives events that occur after it has registered, tuple space can hold data until a consumer requires it. This means in the model who, where and when, none of this information is required prior to communications.

The concept of tuple space has until recently not had much of a following. This is set to change when Sun releases JavaSpaces [15]. JavaSpaces is a full implementation of the tuple space concept and is fully distributed.

This section has reviewed a number of communications technologies. From the start events have been the preferred choice for communication services. The CORBA event bus was reviewed but must be rejected on three counts, one it is centralized, two

it does not support typed events at this time and three the implementations available were to slow. The ECO model is very useful as a template on which to build the event service. Constraints and composite events are not required to test the hypothesis of this dissertation, therefor parts of the ECO model, and the Cambridge model are not relevant.

3.5 Fault Tolerance

Fault Tolerance is an integral part of a worm. With this in mind a number of existing technologies that can be used to provide the worm with varying degrees of fault tolerant behavior are reviewed.

Group Communications

A group is a set of processes that act together as a single entity. One of the major attributes of a group is that a message sent to one member will also be received by all other members of the group. The technique used to deliver messages is normally some form of reliable multicast. The semantics of the message delivery can be determined by the user when the group is created or in some cases, dynamically at runtime. Groups are dynamic, process can join and leave the group without effecting the other members of the group. Tanenbaum [45] gives a good general description of groups and their properties.

ISIS [5] was investigated as a means of building fault tolerance into the worm. As a general paradigm, group communications suits the concept of a worm. The worm as a group with each of its segments as members of the group work well. ISIS has been shown to be heavyweight [19, 4, 37].

Horas [39] was developed from ISIS in response to the problems of flexibility and speed, by allowing the group protocol stack to be modified at runtime. When a new group is created, Horus allows the group protocol stack to be assembled, like a set lego blocks, to suit the needs of the application using the newly defined group.

The ENSEMBLE system [20] takes the concept of assembling a protocol stack

a step further than Horus. The system allows the assembly of a group protocol stack from a set of micro-protocols as Horus does, when a new group is created. ENSEMBLE further allows the protocol stack to dynamically adapt its makeup to suit the network conditions it encounters. If run on a reliable network, ENSEMBLE may remove parts of its reliability protocols to take advantage of performance.

Although ENSEMBLE does address the issues of performance, none of these group protocols have been designed to work in a mobile environment.

This chapter has surveyed a number of technologies with a view to using them to implements parts of the underlying services required by a worm. Migration was surveyed and the Java serialization facilities appear to meet many of the requirements of the worms migration service. Events and especially the ECO model were found to be a good template on which to build an event service. No Java implementations of ECO was available, therefor the event service, although based on the general ideas of ECO, will be built from scratch. Group communications, although ideal for providing the worms fault tolerant services, was found to be too heavy weight for the worms fault tolerant requirements.

Chapter 4

Object Movement

As pointed out in §1.1 migration is an important part of the overall services provided by a worm. The worm has been developed in Java, to take advantage of it heterogeneous environment and its ability to serialize objects. The system implements migration of segments in a worm using a combination of interface specification and Java serialization. The interface design forces the user to implement code that can be migrated transparently using the Java serialization package. Full transparent migration is not possible because Java will not allow threads to be serialized.

4.1 Process Migration

When and for what reason, to migrate a process is normally a policy issue, sometimes that policy is defined at the operating system level, and at other times it may be at the user's discretion. There are numerous reasons why it may be useful to migrate a process and here only a small number are discussed.

Using migration to balance loads in a workstation cluster

If process migration is allowed within a cluster of workstations then it would seem logical to exploit the processing power of the cluster as a whole rather than each workstation within the cluster. To do this the processes running on the cluster should be

spread evenly across all workstations. Spreading the load across a cluster of workstations is know as load balancing[18], and migration is one load balancing technique.

Using migration to conserve network bandwidth

At this point in time the Internet has been growing at an exponential rate and as a direct result of this, there is no longer enough network bandwidth to allow the timely and smooth delivery of data across the network. To alleviate this problem it would be useful to allow code to migrate to the site where the data it requires resides, and process that data locally rather that moving the data across the network processing it and then moving the data back. Database queries and updates as well as forms processing facilities come to mind as examples.

Using migration to monitor events at source

Monitoring real world events at source is another example of where migration can be useful. In a smart building the code that monitors real world events should be moved as close to the event as possible, this will cut down on unnecessary use of network bandwidth. It could be envisaged that if an office is shared then its environmental systems may be controlled by code such as an environmental policy system. This system would detect who is in the office at this time, and migrate the code and state that implemented that user's preferred settings, not unlike the way workstations work at present. When you login then the computer sets up your unique environment.

Migrating a process transparently

The above examples show that using process migration may be a useful policy. The mechanisms involved in moving a running process depend on how well isolated the process is from the underlying operating systems structures. The more integrated a process is within the operating system the more likely it is to have *residual dependencies*,k see [38] A residual dependency may occur when a process that has a temporary file open on a host that does not share a common file system with the target host of

the migration. When the process is migrated then it still requires access to the file of the machine from which it came.

To migrate a process transparently, all access the process has to resources should be encapsulated. The types of resources involved include files, I/O ports memory and the processor. This encapsulation although possible, can prove an expensive solution in terms of processing time, and in many distributed system only certain parts of the system are encapsulated, while other time critical parts such as processor access are not, due to the expediency of execution time over transparency.

Migrating a process that has an active communication channel can be difficult. If a process take 200ms to tear-down move and re-create then what happens to all communications that may be delivered within the migration time. Cahill et al [38] solved this problem with a store and forward mechanism with duplicate detection. The worm also uses this method to handle event delivery during segment migration see §5.5.4.

4.1.1 Code and Data Representation in a heterogeneous environment

A distributed system that functions over a heterogeneous computing environment means the system may be comprised of many smaller networks that may have different topologies and signaling systems. The computers on these networks may come from many different vendors and support multiple processor architectures and operating systems.

Migration in this environment can prove very difficult. First many of the operating system have little or no support for migration. Process code is normally compiled to create machine code, this code only runs on the processor architecture it was compiled for. To migrate a running process between machines of differing architectures requires that one processor is capable of running the other processor's instruction set, normally by using an emulator.

To overcome the processor architecture problem, code needs to be virtualised.

Virtualising code may be achieved by using interpreters created for each architecture, then moving the process as a set of source files. This is how BASIC works. The major difficulty with this approach is slow execution speed.

To overcome the speed limitations some interpreters deal with an intermediate form of code known as p-code. The source files are compiled to p-code, this p-code is processor neutral, but optimized for fast loading and interpretation. Java and some Pascal dialects use this form code virtualisation.

Sometimes the p-code concept is taken one step further by using a Just In Time Compiler to compile the p-code to machine code when loading a code module. This JIT compilation is available in Java 1.2. A small drawback with JIT is the latency experienced when an object is first loaded and compiled. There may be good reasons not to compile all objects in a system, especially those objects that are not time critical or used often.

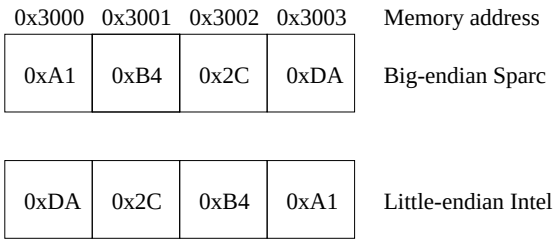
Migrating data like code in a heterogeneous environment is a difficult task. Due to the proliferation of different vendors, many different methods for storing data types have been devised. In this section a brief look at some of the data types and the problems associated with their migration will be reviewed.

The byte or more correctly octet is today the fundamental data type in most computers. Octet is a better description because it is defined as an 8 bit byte. Bytes in general are not required to be 8 bits, and in some small microprocessors can be 4 bit quantities know as nibbles. An octet represents only a number, no coding scheme is implied by this number.

A char is usually an octet that uses the ANSI coding scheme to represent the letters and symbols of the Latin alphabet. It can also use the EBCDIC encoding system created by IBM to represent the Latin alphabet. Software is becoming international and some non Latin alphabets such as Japanese Kanji have more symbols than the 256 allowed by an octet. To overcome this problem a new encoding system known as Unicode [12] has been introduced. Unicode requires two octets to encode a single character and can therefor contain over 65,000 characters in its alphabet.

Integers have a number of differences the first and major problem is the size.

On small machines the default size may be 16 bit but on a Mainframe it can be as large as 128 bits. Integers are stored as a series of octets, how these octets are laid out in memory or the CPU registers becomes an issue. On sun workstations integers are held with the most significant octet at the lowest addressable memory location, this arrangement is known as big endian. Intel based machines hold the the least significant octet in the lowest addressable memory location, this is know as little endian. Figure 4.1 shows how both big and little endian 32bit integers are stored in memory.



Storage of the integer 2,712,939,738 in both big and little endian formats
2,712,939,738 converted to hex is A1B42CDA

Figure 4.1: Big and little endian 32bit integer layouts

Sometimes papers refer to an integer being in Network byte order, this means it is in big endian form, Java uses this form for integer storage on all machines, regardless of their natural endian type.

The final point that requires mention is alignment. Some architectures require that integers are aligned on specific boundaries in memory. One machine type may require integers to be aligned on 16 bit boundaries while another requires alignment on 64 bit boundaries. This alignment often causes problems if complex structures or object are written to file using one alignment and then read back using a different alignment architecture. To align integers within structures or object compilers usually pad the structures or object with octets, see figure 4.2 but when read in this padding makes the in memory structure on the new machine unreadable. This problem will also occur if the structure is passed as a data item across a network connection.

The storage and interpretation of floating point numbers is also subject to

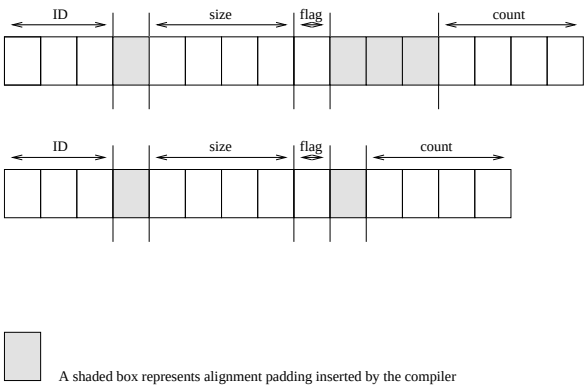


Figure 4.2: Data alignment using 32bit and 16bit boundaries

having many different encoding standards. I do not propose to go into any detail but only to point out that Java regardless of machine type it is running on uses the IEEE 754[1] encoding scheme.

4.1.2 Code granularity and storage

Code granularity and storage is concerned with how executable code, either machine code or p-code is stored ready for execution on machines.

Monolithic

The simplest way to compile code is to create a single machine code image in a disk file. When the user wishes to execute the code then a loader within the operating system will create a process and load the single code file into the computers memory. The code will be executed starting at an entry point defined in header structures that have been placed at the front of the code file.

If memory on a machine is limited this model often uses an overlay system that only load small parts of the code into memory at any one time. Windows NT[43] goes further and treats executables as part of the swap space, loading only the initial code but creating memory pages for all code marking them as swapped out and pointing to their position in the executable file. These are known as memory mapped files.

[1] Java does not follow the IEEE 754 standard rigidly, see [30] for details of Java floating point support

Executables plus shared libraries

Most operating systems now allow parts of the code which may be common to other programs to be moved into shared libraries. A good example would be to move the C runtime code into a shared library. Unix and Microsoft Window's systems use `.so` and `.dll` files respectively for this purpose. When migrating a running process, dependencies on loaded libraries may cause some difficulties. There is no guarantee that the same set of libraries exist on all machine across the network.

Class files

Java was the language used to conduct the experiments in this dissertation, therefor it is useful to understand how Java stores and uses code. Java is an Object Oriented language and all code in Java must exist within a class. Unlike C++ Java does not compile a set of modules into a single machine image. Java compiles all classes into separate class files, even if you define two classes in a single source file it will still output two class files. The Java Virtual Machine deals only at the level of class files. An application in Java consists of a set of class files, one of which must have a static method with the signature;

```
public static void main(String []args);
```

This indicates the entry point for this application to the JVM. The fact that each class is represented as a separate file in Java and that the user can create their own class loaders along with the serialization package, should make object migration possible even without a shared file system.

4.2 Object movement in Java

Before going on to discuss Object movement in Java, we need to be clear what exactly a Java Object is, An Object in Java is an instance of a class, each object is comprised of a set of fields as defined in the class file. These fields, unless defined as static, are unique to each Object. The methods for the object are stored in a Class object which

in turn is created from a class file using a Class loader. An Object in Java has three parts;

1. The in memory Object with its set of fields representing the objects state.
2. The Java Class Object that holds static state for the class as a whole, and the code for each of the class methods.
3. The class file which holds the byte code used by a class loader to define the Class object.

Thread Objects in Java are special, and if an Object is derived from a Thread Object or implements the runnable interface then it will have information about its run state and the address of its stack held directly in the JVM, this information is not normally accessible to the developer. To move a running thread object requires changes to the Java Virtual Machine, as a result the code developed during the course of this dissertation does not allow active Java threads to be migrated. Instead the segment code is structured in such a way that allows the segment to be migrated at know points throughout its execution.

4.2.1 Object serialization in Java

There are, as pointed out earlier, many problems in moving code and data in a heterogeneous environment. If the code developed to test the worm paradigm was not in Java then the best solution to overcome these difficulties would have been to use CORBA. However using Java as the development environment moves much of the data migration difficulties out of the hands of the developer, using the Java Virtual Machine to present a homogeneous view of the working environment across systems on which the JVM has been implemented.

Java has a built in serialization package that allows an Object to be converted to a neutral format byte stream. This stream may be directed to a file to allow persistence, or in our case the stream may be directed across a network. When the stream is unserialized at the destination host, the JVM will rebuild the object using the data

values from the original serialized object. All fields except static and transient types are serialized by default serialization [2]. Java allows the default behavior of these object streams to be overridden by allowing the users to create their own writeObject and readObject methods for ObjectOutputStream and ObjectInputStream classes respectively.

If an object being serialized holds references to other objects in some of its fields then the serialization code builds a hierarchical dependency graph of all objects required, all these object are also serialized with the original object. If you try to serialize an object that does not implement the serializable interface then the JVM throws an exception.

When the ObjectInputStream attempts to unserialise an object, the first piece of information it gets, is the class name of the serialized object. Before proceeding, ObjectInputStream will attempt to load the class using the system class loader. Once the class has been loaded Java can then continue unserialising the object. This process is recursive, objectInputStream recreates the dependency graph for the current object and calls the class loader to load each of the class files that the current object depends on, or has fields containing. This serialization code requires the system class loader have access to the class files of the objects being unserialised, normally these files will be located using the current CLASSPATH setting. If this is not the case then objectInputStream can be sub-classed and the subclass can override the resolveClass method, allowing users to plug in their own class loader[2]. Java cannot serialize a running invocation of an object method. It has no means of saving the method invocation sequence or the local variables. These are stored in the stack of the thread that is making the current invocation. Attempts are being made to save and restore the state of running threads in Java, see Stefen Funfrocken's paper [16].

4.2.2 Class loaders

A Java class file is the finest granularity code module on which the JVM can work. All class files are loaded at run time or when accessed by a previously loaded class

[2]For a more detailed overview of Object Serialization and Java RMI see [1]

within the JVM. Responsibility for loading classes is handled by a class loader. Class loaders can be user defined, but there is always a single system class loader called the null class loader, which is created by the JVM when it starts.
Java Class loaders have four main attributes

1. Lazy loading.
2. Type-safe linkage.
3. User-definable loading policy.
4. Multiple Name-spaces

For a detailed overview of class loaders and these attributes in particular see [41].

The main difficulty with using class loaders is understanding the relationship between class loaders, classes and Objects, especially in relation to the constraints imposed by the JVM to guarantee type safety.

If a class loader L_1 reads in and resolves a class file C_f, it produces a Class object C_o. The class loader L_1 is said to be the defining class loader for the class C_o, and the class is then identified with the fully qualified name(L_1, C_o). If another loader in the system L_2 loads a new version of the class C_f to produce a new C_o then its fully defined name is (L_2, C_o) and both classes although they have the same class name are not treated as equivalent by Java. In fact if you attempt to cast an object defined by (L_1, C_o) to a reference defined by (L_2, C_o) then Java will throw a class cast exception. This is how Java class loaders implement multiple name-spaces, figure 4.3 shows the relationship between class loaders and name-spaces graphically

To allow classes from different name-spaces to be addressed, Java allows the use of interfaces. If classes implement a common interface and this interface is loaded by a class loader that is the parent of both L_1 and L_2 then objects from both classes may be addressed by casting them to a reference type of the common interface, and using the interface methods on the objects.

Another method that can be used for addressing across names-spaces is reflection, details of using the Reflection API can be found in [29]. Reflection in Java allows

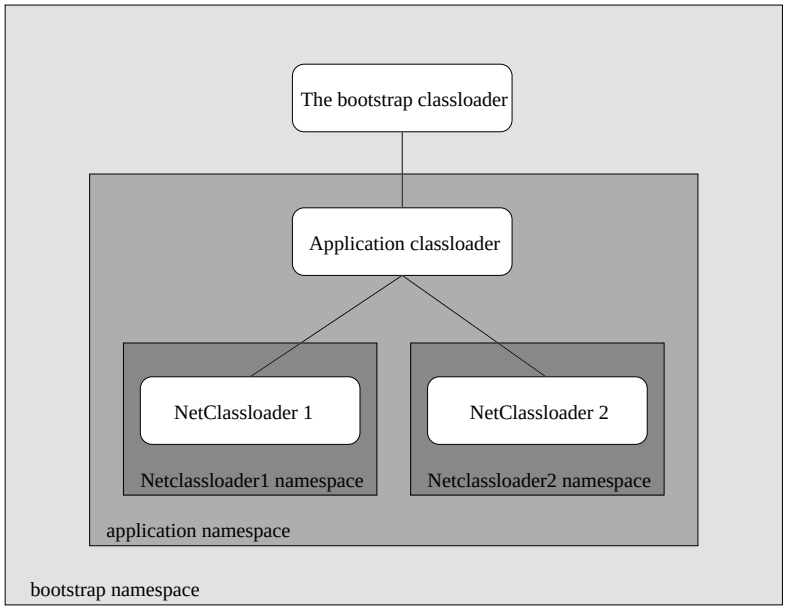


Figure 4.3: A graphical view of class loader enforced namespaces

us to ask an object for its class and then ask the class to enumerate all methods and their signatures. Armed with a method name and signature, it is possible to use the Java reflection API to make calls on objects that cross name-space barriers.

4.3 Security issues

The dissertation does not deal with security, but in a commercial environment, security would be a major factor. Java 1.2 does allow the developer to implement a reasonably fine grain security policy that a real environment for mobile objects requires.

This chapter has discussed the problems and implications of moving code and data in a heterogeneous environment. It has shown that migration is a difficult task, and in particular transparent migration is very difficult. It then went on to look at the structure of code and data in the Java language and how Java serialization and class loading work. By using serialization, objects can be migrated, add to this the ability to move the class files using a class loader, code may also be migrated. This in combination with forcing the user to implement their code using a special interface

allows the general user process to be migrated seamlessly at runtime.

Chapter 5

Analysis and Design

This chapter brings together the information gathered from the previous two chapters, to layout the design for a mobile worm. The Objective of this chapter is to produce a design that will lead to the implementation of a worm with the following characteristics.

1. It can live in a heterogeneous environment.
2. The segments can host user computations.
3. The segments can be migrated by the host server, without effecting the user code.
4. The segments can communicate with each other regardless of location.
5. The worm must be capable of replacing failed segments.
6. The management of the servers and worms should be distributed.

The design falls into five broad categories, the host server, the migration service, the event service, the worm, and finally the user applications used to test the viability of the system.

5.1 The target environment

The target environment of any new system plays a major role in the tools used to create that system, the design characteristics are also dictated by the target environment and by the requirements of the systems users.

5.1.1 Heterogeneity

From the outset it has been envisaged that the worm should work across a range of environments. Bearing in mind the difficulties exposed, developing for a heterogeneous environment, it becomes prudent to offload as much of the house keeping code to an all ready functional system. The choice of Java as the development language allows much of the burden of heterogeneity to be borne by the Java Virtual Machine. The JVM allows us to move data and code in the form of objects and classes across any platform on which the JVM has been ported. Figure 5.1 shows how the Java Virtual machine shields Java programs from the underlying operating system architecture.

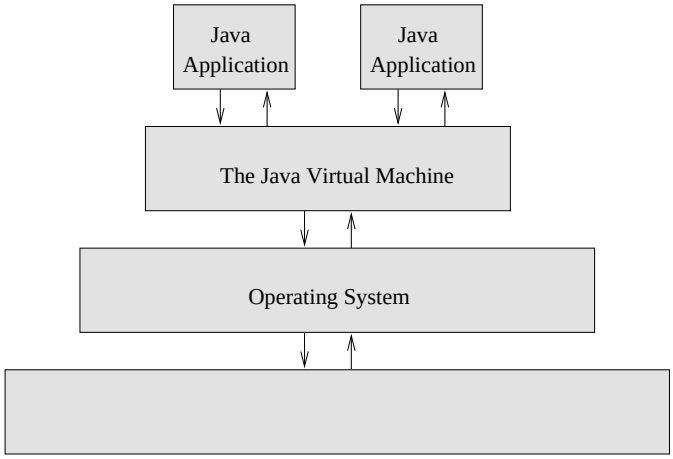


Figure 5.1: The server GUI

5.1.2 Dispersal

The worm should run across a number of machines, the scope of the network should not be limited to a set of machines that share a common file system. In theory

the worm should be capable of running on a subset of nodes running anywhere on the Internet. In practice this environment all-though feasible, requires the use of IP Multicasting and in particular access to the MBONE. Each segment once dispersed should exhibit no residual dependencies on its original parent node. This is one of the key differences from Mathews worm implementation. The segments should be capable of being migrated as many times as is required by their parent nodes to keep loads balanced within the virtual network.

5.1.3 Mobility

As a result of the load balancing, and fault tolerance requirements, the communications protocols must be able to cope with a very dynamic environment. The segments and the user computations must be capable of being moved without loss of function. It is desirable to make the migration process as transparent as possible, but without making substantial modifications to the JVM, full migration transparency is not possible. Migration is achieved by imposing an interface that all user code must implement. The interface imposes a well defined structure on code. This structure is designed to be similar but not identical to the interface used by the Java applet class.

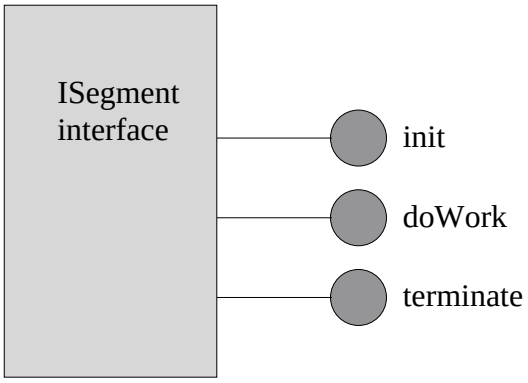


Figure 5.2: A partial view of the ISegment interface

The interface has three main method calls concerned with the general running and migration of the workers, shown in figure 5.2. Init is called when the worker is first instantiated.The doWork method is called on a continual basis to allow the worker to do work. When a worker returns from this call with a return code of true, it

is indicating to the host that it has more work to do, This means the host environment will continue to call the segments doWork method. When the segment processes a return from a doWork call, it is allowed to migrate the worker. It is the responsibility of the worker to break its workload into small chunks, each of which will be completed within a single iteration of the doWork method. Smaller work chunks leads to more calls to doWork, this in turn creates a finer grain migration facility.

5.2 The host architecture

An overview of the main components of the server architecture are shown in figure 5.3

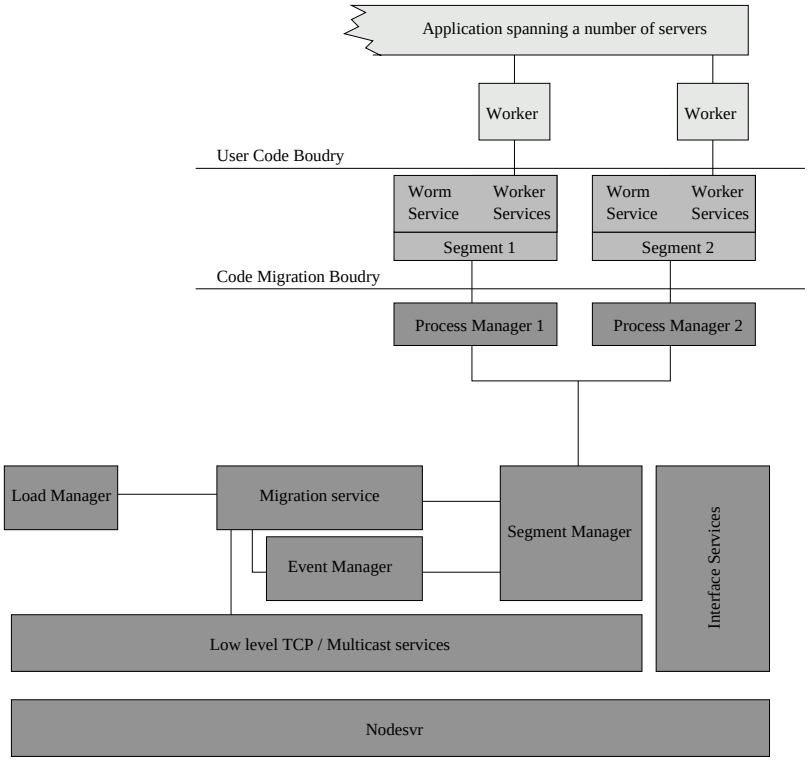


Figure 5.3: The worms working environment

The server hosts five major services:

- Segment Management controls and hosts worm segments. It should maintain a safe and separate working environment to host worm segments.

- Migration Services control the movement of Java objects between servers.
- The Event Manager is responsible for implementing a fully distributed event service.
- The Load Manager monitors the load on the server, and if the load exceeds a user define high water-mark, it is the responsibility of this module to find a new host with a lower load factor, and to initiate the migration of one of the hosted segments to the new host.
- The Graphical User Interface module is used to allow the user to control some of the internal aspects of the server. In particular it also allows the user to move or kill currently hosted segments.

5.3 Segment hosting

Each segment is hosted within its own environment. Each of these environments run in their own separate thread. Running these environments as separate threads allows us to exploit concurrency in the servers.

A server should be capable of running segments from more than one worm at the same time. If there are a number of segments, making up separate worms, each developed independently, running on the same server, at the same time, problems arise when two segments create and use objects with the same name, but different functionality. This is a name-space problem, to solve it, requires a separate name space context for each running segment. Java, as we have seen in chapter 3 enforces name-spaces using class loaders. Using a separate class loader for each segment enforces the separate name-space requirement for each segment. The class loader holds the class objects and also the class bytes, loaded from the class files for all the segments loaded classes. This ensures that when segments are migrated the destination server can obtain a copy of the class bytes directly from the source host, and there is no dependency on the original node where the segment was loaded. Figure 5.4 shows the segment hosting environment and the division of name-spaces within it.

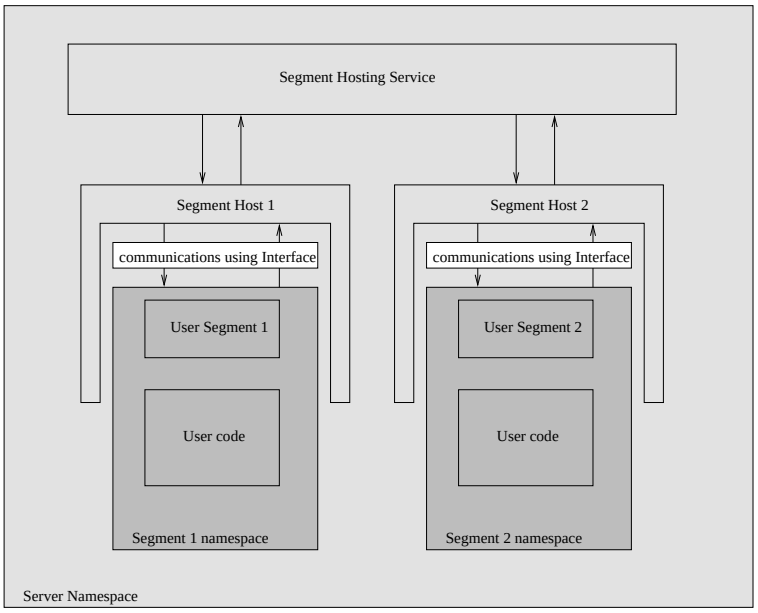


Figure 5.4: The Segment Hosting Environment

Having separate name spaces for each segment introduces problems of addressing. How can the segment, or the server, cross name-spaces if they wish to communicate. This is done in Java using interfaces. As pointed out earlier the name-space model in Java imposes a hierarchical trust chain. To allow objects from separate name-spaces to communicate, objects must implement a common interface. This interface must be loaded by a class loader that has a parental relationship to the class loaders of the objects that wish to communicate.

The following interfaces have been defined in the system to allow communication across name-space boundaries:

- ISegment as we have seen earlier is used so that the segment host environment has a means of making method calls to the user supplied segment code.
- ISegmentInternal is used so that the segment host environment can communicate directly with the internal segment code.
- INodesvr is used by the segment to communicate with the server.
- IEventManager is used by the segment to communicate with the Event Manager.

- IEvent is used by both the Event Manager and the segment to handle events as they pass through the system.

5.4 Migration Services

It is the job of the Migration service to move running segments between hosts. The service will avail of Java Object serialization to move segments across the network. The Migration Service will not attempt to move Objects inherited from java.language.Thread. The move protocol is simple, and is always initiated by the source host. The source first attempts to locate a suitable destination host by calling on the load manger to return the lowest loaded host on the network.

Future versions may allow a named host to be requested from the load manager, if the named host is on the virtual network and its load is less than the user defined high water mark for that server then the load manager should return the requested host in preference to the host with the lowest load. This can be useful for user segments that are being used to monitor or provide services at a particular host.

The Migration service will work in conjunction with the segment hosting service to enable the transparent migration of a segment and the setup of the segments new host environment, including the creation of the host environments class loader. This crossover is necessary because the Java serialization API requires that the class loader be initialized and ready before an object can be unserialized. Figure 5.5 shows migration protocol interaction between two servers on a time line.

5.5 The Event Manager

The Event Manager provides all communication services for segments and the user programs they host. Creating a fully distributed event service is the main criteria governing the design of the Event Manager. No single node within the virtual network has responsibility for managing the event service. The event service is based on the push model. This model is designed so the event producer is responsible for the

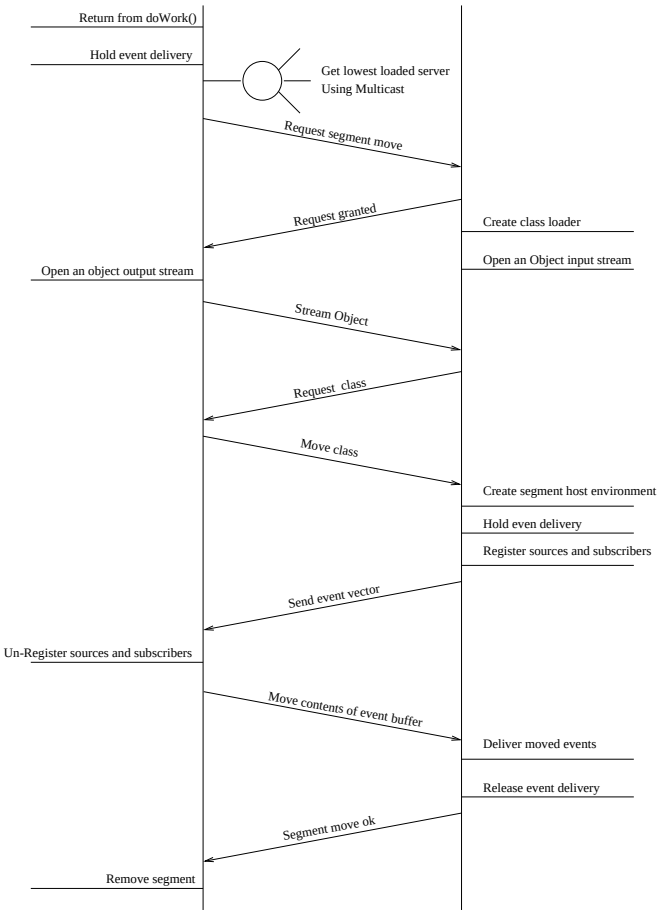


Figure 5.5: The Segment migration protocol

initiation of event propagation. Events may be any Java Object derived from the CEvent object, they must also implement the IEvent interface. This means that events are user defined and fully typed. No event ordering is enforced by the system.

The event delivery mechanism may appear to deliver events, in the order that they are sent from a particular host. This would be true if sources did not migrate. Allowing a source to migrate can lead to the following situation. If a source is located on a heavily loaded server where many segments reside, and these segments are generating events, the dispatch buffer will be very busy. If a segment is migrated to a new server with no load then the next events it generates after migration may well arrive before some of the events that it generated on its original server. These old events will still be queued up in the dispatch buffer of its original host.

The Manager will have three main functions;

1. Registering and Unregistering Sources

2. Registering and Unregistering Subscribers.

3. Raising and Delivering events.

Figure 5.6 gives an overview of the important structures contained within the event manager.

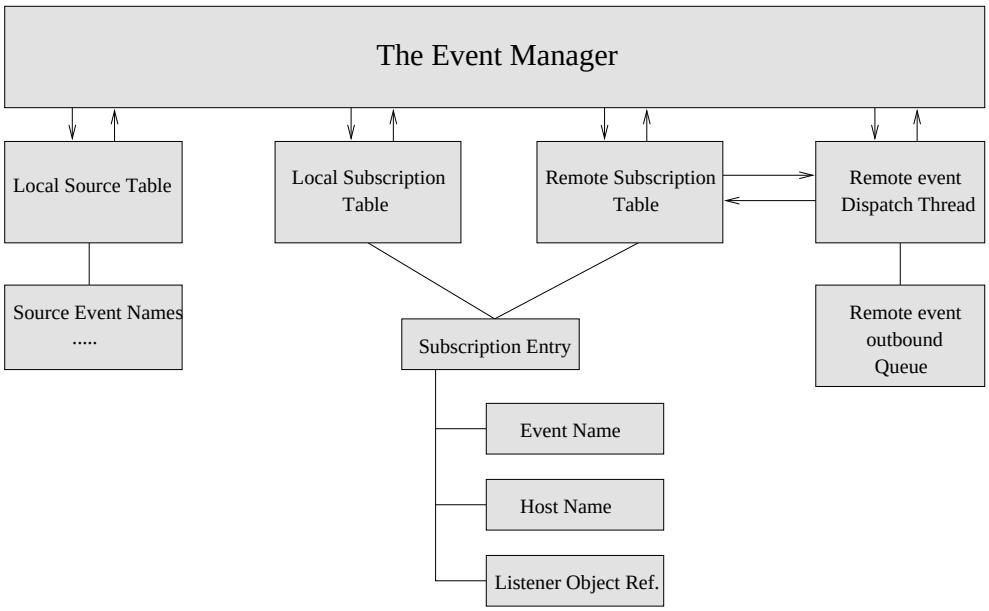


Figure 5.6: Event Manager structural overview

5.5.1 Registering and Unregistering Sources

Registering a source

To register a segment as an event source the the user code will make a call to a RegisterSource method, defined in the IEventManager interface. The Event manager keeps a table of currently registered sources as part of its internal data structures. If this is the first source for this event on the host then a new source object will be created. The source object is then entered into the Event Managers local source table. The source object is responsible for keeping all information about a source, and its subscribers for a particular event type. If this is not the first source to be

registered on this host then an *active sources variable* within the event object will be incremented. Events are registered by passing into RegisterSource the events class-name. When a new event source is registered the event manager must propagate this knowledge throughout the virtual network. Propagation is done using multicasting. The new source sends information about its location, and the event name that it is a source for, onto a known multicast port all servers listen to. When a server's Event Manager receives a new source message it checks its own internal local subscriber table for any subscribers that have already subscribed to this named event. If there are any subscribers in the Event Managers local subscriber table, it will open a TCP connection to the new source's host and request that the subscribers server be added to the new sources subscription list. If there is more than one subscriber registered only one entry will be entered into the sources subscription table. This entry also holds the number of active subscribers on this server.

To summarize, when a new source is registered, it is entered into the Event Managers source table, the event manager publishes the new sources existence to the virtual network using multicasting. Any servers that have active subscribers to the event will contact the new sources Event Manager, registering their interest in receiving events from the new source. The Event Manage will create a new source table and enter the address of all interested parties into it.

Unregistering a source

Unregistering a source involves decrementing the *active sources variable*. If this variable is 0 after it has been decremented, this indicates to the Event Manager that there are no more active sources, as a result the Event Manager removes this source entry from the local source table and tears down any threads associated with the source object.

5.5.2 Registering and Unregistering Subscribers

Subscribing to an event

Subscribing to an event is done by the user calling the Subscribe method defined in the IEventManager interface. The method expects the class name of the event the user wishes to subscribe to, and a reference to an Event listener object that implements the IEventProcessor interface. The EventManager enters the new subscriber into its local subscriber table and then multicasts a locate event sources across the virtual network. When a host receives a locate event multicast it passes the request onto the Event Manager. The Event Manager checks its local source table. If there is a source for this event in the table the subscribers host name will be added to that sources remote subscriber table.

Unsubscribing to an event

To Unsubscribe to an event the user calls the Unsubscribe method defined in the IEventManager interface. This method removes the subscriber from the Local subscriber table. The method then checks if the removed subscriber was a remote subscriber, if so it reviews the remote subscriber table to find out if there are any other subscribers for this event. If the current subscriber is the last remote subscriber for this event then the Event Manager multicast an Unsubscribe event message. A server that receives the Unsubscribe event multicast checks its remote subscriber table in the source for the event name passed in the multicast message, if the host is in the remote subscriber table then it will be removed.

5.5.3 Raising and Delivering events

A user can raise an event by calling the RaiseEvent method defined in the IEventManager interface, and passing a single parameter, a reference to a user defined event object. The Event Manager passes the event object onto the source object for this class of event.

Delivering a local event

The source object will first check its local subscriber table, if there are local subscribers it will process each subscription in the following manner;

1. The event object is serialized to an in memory byte stream.
2. The byte stream is unserialized back into an event object using the subscribers class loader. This allows the object to cross a name-space boundary.
3. The event is pushed onto a delivery buffer located in the subscribers segment.
4. The segment has an event dispatcher thread that continually waits on events becoming available in the delivery buffer. When an event becomes available it is delivered by the dispatcher to the subscribers listener object. The listener object implements the IEventProcessor interface which has one method processEvent. This method take a single CEvent object parameter.

Note that events must be serialized because they cross name-space boundaries. When an event is raised it is within the name-space of the current source segment but when it is delivered it must be in the name-space of the subscriber. The Event Manager uses the IEvent interface, which all event object must implement, to communicate with an event as it passes through the system.

Delivering a remote event

When the source object has processed all its local subscribers, it will then iterate through its remote subscriber table. For each entry in the table, it will enter the event object plus its destination host into a delivery buffer located in the event manager. This remote delivery buffer has a thread associated with it, whose job is to wait until there are entries in the buffer. When an entry becomes available, the thread will use the migration service to deliver a single copy of the event to the required host.

When the destination host receives a new remote event, the server will will call the DeliverRemoteEvent method in the Event Manager. This method delivers the event in the same manner as an event is delivered to a local subscriber.

5.5.4 Event Delivery and migration

The delivery buffer resides in the segment. This arrangement allows easier migration. When a segment migrates, its buffer remains active during migration, but has been instructed not to deliver any events to listeners. When the migrated segment is up and running on its new host, the new segment re-registers all its sources and re-subscribes all its subscribers. It will not activate event delivery, so any events delivered to the new segment will be held in the delivery buffer. The new segments now sends back confirmation that migration is now complete. The old segment will Unregister all sources and UnSubscribe all its subscribers. The old segments final job is to pack up all the event in the delivery buffer into and vector and move this vector across to the new segment. The new segment will deliver these events directly to the appropriate listeners. Once all old events have been delivered, the new segments delivery buffer is allowed to start delivering events as normal. The old events name, source host and number are held in a vector, each new event delivered will check this vector to see if the event it is about to deliver is a duplicate. If so it will be discarded.

5.6 The Load Manager

The implementation of load allocation on the server is rudimentary and was designed for use primarily as a testing tool that can be used to force segment migration. The algorithm is based in part on a partial knowledge threshold algorithm found in[18].

Load as defined in the server equates to the number of worm segments that are currently being hosted on the server. The load manager works using four key variables;

1. The current number of segments on the server.
2. The maximum number of segments on the server.
3. The segment hi-water mark. This is set when the server is loaded. if the current number of segments exceed the hi-water mark the load manager will attempt to migrate an arbitrary segment.

4. The bias value. Having made a decision to migrate a segment, the server will request load values from other servers on the virtual network. It does this by multicasting a load request message and the waiting a preordained timeout to assemble the replies into a load vector for all servers on the network. It should be noted that due to the nature of multicast all servers may not reply. Once the timer expires the load manager takes the lowest value from the load vector adds the bias variable to it and only migrates a segment if this value is less than the current load on this server. The bias value prevents a ping pong effect from occurring between two servers that both just exceed their hi-water mark.

The user can prevent their segment from being migrated by setting a don't migrate flag in the segment. This is useful if the segments workload is nearly finished and migrating would waste processor time and network bandwidth.

The load manager has its own thread that will periodically monitor the load on its server. It will also have a public method that when called will return a suitable host for the target of a migration request.

5.7 The GUI

The Graphical User Interface implemented by the server is used primarily for debugging and testing. In a production version the user should be able to monitor and control the state of any server from any host on the virtual machine. the current interface displays the name of all segments hosted on the server. The interface will also display any exceptions or messages generated by a running server. The interface allows the user to perform four main tasks.

1. By hi-lighting a segment in the currently hosted segments area the user can force that segment to be migrated pressing the *Move Segment* button.

2. The user can also kill the hi-lighted segment by pressing the *Kill Segment* button.

3. By pressing the *Display Threads* button the user can open a window that allows the display of all currently running threads. All threads in the server have English names that explain their use.

4. Pressing the *Terminate* button will terminate the server.

5.8 Runtime configuration

This section covers miscellaneous parts of the server design that do not belong to any discrete service.

5.8.1 Policy Management

The server has a number of key variables that control its operations. All these variables have default values that are initialized when the server is first started. The policy manager can override these variables by reading in user defined values from a file called *Nodesvr.conf*. Examples of entries from this file are shown in figure 5.7.

5.8.2 The threading model

Java provides two threading libraries [36], Native and Green. Native threads as the name implies directly map the Java threads to the threading model of the host operating system. Obviously each system has its own threading model and scheduling algorithms. If the design takes advantage of some of these features then the correct operation of the servers internal threading model becomes complex and OS dependent. Green threads are a generic thread model designed and implemented by Javasoft [23]. This is a non preemptive priority based threading model.

5.8.3 Scheduling Threads

The system has been designed to run on the green thread model. This allows it to be used consistently across all OS platforms supporting Java. Using the green thread model does not prevent the user from running the system under the native thread

```
# This is the Nodesvr configuration file
#
# Bob Ashmore
#
# 8th June 1999
#
PORT = 5050
MAX_LOAD = 10
HR_TIMEOUT = 1000
MulticastPort = 9000
MulticastGroup = 225.4.5.15
MulticastTTL = 20
MAX_SEGMENTS = 10
HIWATER_SEGMENTS = 7
LOAD_BIAS = 2
LOADCHK_INTERVAL = 20000
LOADREQUEST_TIMEOUT = 1000
SOURCE_EVENT_QUEUE_SIZE = 500
EVENT_BALKING = 0
IGNORE_LOAD = 1
```

Figure 5.7: An example configuration file for the server

model if they wish. The important consideration, when using the green thread model, is to make sure that the systems threads all obtain reasonable access to the processor. In designing the servers threading model it is important that the scheduling of all threads running network communications should be prioritized so they always run to completion without interruption. This prevents communications timeouts or thread preemption at the destination endpoint due to preemption of the thread running at the source endpoint. To ensure this, server threads are run at two separate priorities. Threads that process network communications are run at the higher priority, while all other threads run at normal priority.

5.8.4 Debugging

The server will have a method to log information to disk, this method will be called at all critical sections as the server processes work. This log will be useful when

debugging and testing the system. The log will also include timing information.

One of the GUI buttons will allow the display of currently running threads. This is also useful when debugging the system.

5.9 The Worm

This section covers the design of the worm. The worm is really only a logical entity, it exists as a set of running segments, and has no centralized control structures.

5.9.1 Naming requirements

Because the servers can host a number of worms simultaneously, each worm must have some form of distinguishing factor. A worm in this system uses its name as a unique identifier. When the worm is created, all segments are instantiated on the same host, then dispersed throughout the virtual network. The host on which the worm is created must be running a server. The creation utility will ask the server to create a unique name. The server will generate a name using the hostname and appending a sequential number to it, which it reads from a special file. The number is incremented and then written back to the file. All this will be done in a synchronized method to make sure that the file is only accessed by one process at a time.

5.9.2 Creating the worm

To create a worm a user creates there own Java program which will instantiate a Worm creation object, passing in the total number of segments the worm will require. The user will then instantiate the segment code and send each new segment to the worm creation object. When the worm has all the segments it will dispatch each segment into the virtual network.

5.9.3 Monitoring life across the worm

The first problem is how to define death. Because segments are dispersed across the virtual network, the segments that make up the worm, can only ever have partial knowledge of each others state §2.5. In this system death will be defined by the absence of communication for a reasonable time. The time element will be user definable. One way of monitoring segments is to have all segments in a worm monitor all other segments. This works, but when a segment dies all other segments must come to consensus that this is so. Having achieved this distributed consensus, one segment must be elected to re-create the dead segment. This type of algorithm is overly complex for what is required in this system.

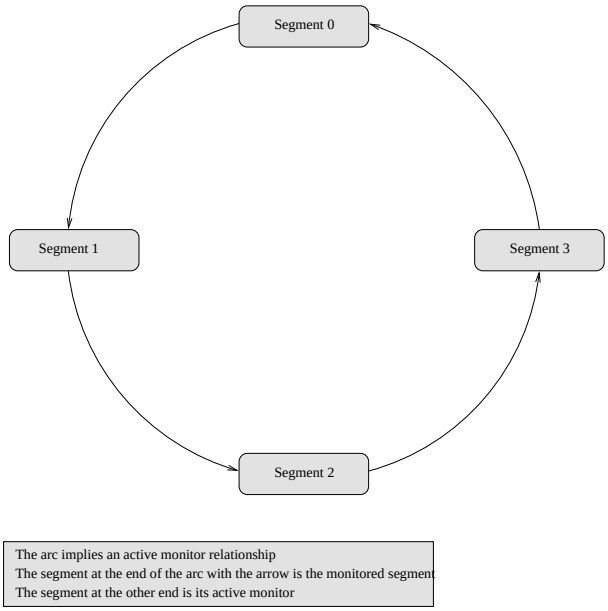


Figure 5.8: Segment Monitoring

A simple ring algorithm will be used by the system. All segments when created will know how many segments are in the worm. Each segment will be assigned a segment and generation number on creation. Taking as an example, a worm that has four segments, segment 0 monitors segment 1, segment 1 monitors segment 2 and segment 2 monitors segment 3, finally segment 3 monitors segment 0. This arrangement is shown in figure 5.8. If segment 0 monitors segment 1 then segment 0 is known as segment 1's active monitor. Monitoring will be done using a heartbeat

event. Each segment will issue a heartbeat at a regular user defined interval. The monitor listens for this heartbeat using a timer to check the interval. If the monitor fails to receive more than x heartbeats, where x is also user defined, then the monitor will pronounce the monitored segment dead. It can then create a new segment with a generation number greater than the one it was originally monitoring, and send it out into the virtual network.

5.9.4 Life after death

The definition of segment death is imprecise and based on partial knowledge. The case can occur that a dead segment mysteriously re-appears in the worm. To cope with this situation, each segment when it is created is given a generation number, starting with 0. Contained in the heartbeat event is the generation number of the sender. If a segment receives a heartbeat event with the same segment number as itself, and a later generation number, then the receiving segment must terminate.

5.9.5 Re-creating dead segments

It is the job of the active monitor to re-create a segment if the segment is is monitoring appears to have died.

Homogeneous segments

Re-creating a segment that is identical to its monitor involves a simple procedure of instantiating a new segment using the current segments class, and then asking the server to disperse the new segment.

Heterogeneous segments

When the segment being monitored is not derived from the same class as its active monitor, then the active monitor must have access to the monitored segments class file. In this system the first heartbeat will pass the home server of the monitored

segment. This will allow the monitor to request the class file from the home server and then proceed as before.

This chapter has covered the basic design of the worm and all its services. A great deal of emphasis has been placed on the design of the event service as this is critical to the development and testing of the hypothesis, are events a suitability communications paradigm for implementing a worm in a mobile environment, put forward in this dissertation.

Chapter 6

Implementation

Much of this chapter is straight forward, the structure follows closely on that of the design chapter.

6.1 The Server implementation

The server is implemented by the Nodesvr class. This class has the required Java entry point *public static void main(String [] args)*. Nodesvr is, as the design implies, a host class for all the services available to the worm. Nodesvr has a number of methods that allow the worm access to the hosted services. There are also some small utility methods used internally by itself and other services.

6.1.1 The communications handlers

The most important function Nodesvr performs, is the implementation of the communications protocols. Originally Nodesvr used a thread per message paradigm [28]. This was implemented by a ProcessClient object. The design worked well when the number of messages between servers was small. When the Event Manager was brought on-line the number of inter server messages grew to such an extent that the server would run out of thread resources, because each method was being handled by its own thread. Threads take time to instantiate and also use resources.

The current design has four process threads, each listening on a separate port, and each responsible for handling separate message types. This design works, but is not optimal, the best approach would involve the implementation of a thread pool [36] to handle communications. Each time a new message arrived at the server, one of the threads from the pool would be assigned to handle the message. If all threads in the pool are busy then the message handler will block until it can acquire a handler thread from the pool. This design puts an upper bound on the number of threads the server can create, so avoiding resource exhaustion. In the current design there are four distinct message handlers;

1. Object migration.
2. Class migration.
3. Event migration.
4. Load requests.

Each handler is implemented using a portMonitor class. This class is derived from the Java Thread class. These message handlers are instantiated in the Nodesvr's constructor. Each handler opens a server socket on the port passed into it's constructor, and obtained from the policy manager. When a message arrives on the monitored port the handler calls a general Nodesvr method processRequest. The processRequest method is responsible for processing all inter server communications. This method implements the inter server protocols, these protocols all run over TCP. The inter server protocols are normally of the form *command:parameter* and processMethod will parse the text message into a command and its parameters as separate strings. It then process each command using a set of *if else* statements, passing on control to the appropriate service as specified by the command. The processRequest method implements the following communication commands;

- Request new worm id. This message is generated by the worm creation object CCreateWorm, it is used to request a unique name from the server for a newly created worm.

- Request segment create. This message is also generated by the worm creation object. It is a request by the creation object asking this server to host a new segment.

- Request segment disperse. This message can be generated by the worm creation object, or the internal heartbeat object, it is requesting that this server immediately move the incoming segment to a new host. This is how a worm is normally dispersed across the virtual network from the server on which it has been created.

- Request segment move. Segment move is the message sent by a server wishing to move a segment to the server that receives this message.

- Request event move. Event move is sent by a server to this server to initiate the moving of an event to this server.

- New remote subscriber. This message is sent by a server's Event Manager, in response to a new source being registered on this server. It returns information about subscribers on other servers that wish to be added to the new sources subscription lists.

- Request class. This message is sent by a netClassLoader on another server, to request a class file, resident on this server be sent.

- Load request. This message is sent by other servers in response to a load request multicast by this server. It contains load information from the sending server.

There are a number of helper classes that automate the task of sending command strings across TCP streams easier. There is also a method in the server called broadcast which uses a multicast socket to broadcast a text string to all servers on the virtual network.

6.1.2 The server interfaces

The server implements a single interface, INodesvr,see figure 6.1 that allows the segment which resides in a separate name-space to communicate with it. Most of the interface is a wrapper for the IEventManager interface so the segment only has one point of call to the server, and all its services. The only other method dontMoveMe is used by the segment to inform the server not to move the segment.

```
public interface INodesvr {
  public void RegisterSource(String name);
  public void UnRegisterSource(String name);
  public void Subscribe(String name,Object subscriber);
  public void UnSubscribe(String name,Object subscriber);
  public void RaiseEvent(Object evt);
  public void dontMoveMe(boolean flag);
}
```

Figure 6.1: The server Interface, INodesvr

6.2 Segment Management

All segments are hosted in, and controlled by CProcessUserObject. This class is derived from the Java Thread class. When a new segment arrives at the server in response to a *Request segment create* message, the server creates a new CProcessUserObject and passes the segment reference of the newly unserialized segment into this new object. The server keeps a table of CProcessUserObjects called userObjectTable, when a new CProcessUserObject is created it is added to this table. When a segment terminates or is moved, the server removes its CProcessUserObject entry from userObjectTable.

The CProcessUserObject acts as a host environment for the segment. The segment and its class loader references are held in CProcessUserObject using a wrapper class, CSWrapper. The wrapper can be retrieved by the event service, which requires both the segment and its class loader, to make the inter name-space transfer of an

event object destined for the hosted segment.

Each CProcessUserObject also holds an event dispatcher that queues and delivers events to all subscribers in this segment. The queue resides at this level, rather than in the event manager, to assist in segment migration.

The worms heartbeat mechanism is also hosted by the CProcessUserObject. This ensures all worm management functionality is located at the CProcessUserObject level.

6.3 The Migration Service implementation

Migration is achieved using Java object serialization and class loading. There are three main classes that control migration in the system.

1. The class loader NetClassLoader
2. The class provider NetClassProvider
3. The overridden objectInputStream NetClassObjectInputStream

The NetClassLoader, derived from the Java ClassLoader class is used to load classes across the virtual network. A new NetClassLoader is created for each segment that is moved to a server. This ensures that all segments reside in their own namespace. The segment loader is called by the NetClassObjectInputStream to resolve any objects that are being UN-serialized across the network. If the loader cannot find the class for the newly UN-serialized object, by first attempting to delegate to the application class loader, and then if this fails, its own class cache, it will send a *request class* message to the source server. The source servers processClient method delegates the handling of this message to a NetClassProvider object. This object will locate the class, either as an array of bytes in the loader of the object just moved, or as a class file in the servers current directory. NetClassProvider ships classes as an array of bytes, exactly as read from a class file. These class bytes are moved to the new loader.

Each loader keeps two forms of each class it loads, the first is just an array of bytes identical to a class file image. The second is as a Java Class object used by Java to instantiate new objects of this class. The bytes are held so there is no dependency on a home node for any class associated with a segment. If an object is created on A, moved to B and finally moved to C, keeping the class bytes in the class loader allows us to move the object any number of times requiring only access to the node from which it has just been migrated, for all its class and object information.

Another reason why each segment gets its own class loader, is to ensure no class files are left on the machine after a segment has migrated. Class files are only removed from the JVM when the class loader for those classes has been garbage collected. When the segment has been migrated the reference to its class loader is removed so allowing all trace of the segment, its objects and classes to be garbage collected by the JVM.

Events are also moved using Java serialization. The Event manager has its own NetClassLoader which is uses to load user defined event classes that it receives from other servers.

6.4 The Event Service

This section covers the event service implementation. The implementation follows closely the design laid out in the design chapter. The section starts with an overview of the event manager as seen by a user of the system. It then goes on to describe briefly the main classes that make up the event service. Finally it steps through the implementation of each of the main services the event manager provides.

6.4.1 CEvent, "The mother of all events"

The class CEvent is the base class for all events within the system. It is a simple class that implements the Java serializable interface. This allows CEvent, and all classes derived from it, to take advantage of Java serialization and therefor allow events to be moved using the migration service. CEvent also implements the IEvent interface

which will be covered in the next section. CEvent contains four fields most of which are used by the event service to manage event flow.

1. *name* This field is not used by the event service, but can be set by the user for their own purpose.

2. *process* This field is set to the name of the worm before the event is sent.

3. host This field is set to the host name from where the event has been sent. If this field is null then the event is local.

4. *sequenceNo* This field is set by the segment sending the event, each event as it is raised, is assigned a unique sequence number.

5. *SegmentNo* This field is set to the segment number assigned to the sending segment when it was created.

Events are typed, that is the user defines their own events by deriving new classes from CEvent. The event manager identifies events by their class name. When registering a new source, or subscriber, one of the required parameters will be the class name of the event.

6.4.2 The event manager interfaces

The event manager exposes three interfaces to the word. The user when communicating with any part of the event service, or indeed with the event itself, will use one of these interfaces.

IEvent

Figure 6.2 shows the IEvent interface, this interface is used to access all the core fields in the CEvent class. It allows access to an event as it passes through the system, even though the event may not be in the name-space of the event manager.

IEvent as seen from Figure 6.2 allows the user access to the internal fields of the CEvent class.

```
public interface IEvent {
  public void setName(String nme);
  public String getName();
  public void setHost(String hst);
  public String getHost();
  public void setSequence(long seq);
  public long getSequence();
  public int getSegmentNo();
  public void setSegmentNo(int sno);
  public void setProcess(String pro);
  public String getProcess();
}
```

Figure 6.2: The event Interface, IEvent

IEventProcessor

The IEventProcessor must be defined by a user class that will act as a listener object for event subscriptions. Figure 6.3 shows there is only one method, processEvent. The user must implement this method on each of its subscription listeners. The event manager will deliver events by calling the processEvent method from the subscription object registered with the event manager,to handle this event type.

```
public interface IEventProcessor {
  void processEvent(Object evt);
}
```

Figure 6.3: The Event Manager Interface, IEventProcessor

IEventManager

The IEventManager interface is the users access to the event managers services. It allows the user to perform all the normal functions an event service provides.

The name field passed in by all methods is the class name of the event concerned. The subscriber field is a user object that implements the IEventProcessor

```
public interface IEventManager {
  public void RegisterSource(String name);
  public void UnRegisterSource(String name);
  public void Subscribe(String name,Object subscriber);
  public void UnSubscribe(String name,Object subscriber);
  public void RaiseEvent(Object evt);
}
```

Figure 6.4: The Event Manager Interface, IEventManager

interface. This object's processEvent method will be called by the event manager to deliver any instances of the event to the this subscriber.

6.4.3 CEventManager The core service

The core of the Event service is implemented in the CEventManager class. This class holds all the key tables that control both the sources and subscribers. All local subscribers are held in the LocalSubscriberTable. Remote subscribers are held in the counterpart RemoteSubscriberTable. A table of the current sources on this server is maintained in the LocalSourceTable. The other two important objects resident in the CEventManager are the class loader used to UN-serialize events and the RemoteDispatcher thread.

The Remote Dispatcher

The RemoteDispatcher thread is responsible for queuing and dispatching all events from sources on this server, destined for other servers. Event delivery is uncoupled from both the sender and the subscriber by synchronized bounded buffers. These buffers have their own dispatcher threads, responsible for event delivery both locally and remotely. The size of these event buffers is user definable in the *Nodesvr.conf* file. De-coupling event delivery smoothes out the operations of the event manager, allowing producers and consumers to maintain an even flow during busy periods.

Keeping state on event sources

Each new event source type registered is managed by a CEventSource object. If there are two sources, both producing the same event type, both will be managed by one CEventSource. Reference counting is used to indicate when there are no more sources left and the CEventSource can be discarded. All events raised will be processed by their event source. The source will lookup the local and remote subscriber tables in the event manager to locate where to deliver the event. Local events will be delivered locally and the event source is responsible for ensuring that events cross name-space boundaries correctly.

Keeping state on event subscribers

Information about subscribers is maintained in a wrapper class CEventSink. CEventSink holds the event class name, the subscribers host name and the subscribers listener object. These Wrapper classes make up the entries in both the local and remote subscriber tables. Local subscribers will have the host field set to null and the listener field set to their listener object, whereas remote subscribers will have their host field set to the host on which they are currently residing, and their listener objects set to null.

The local subscriber table contains entries for all subscribers that have subscribed to any event, local or remote. The remote subscriber table contains entries for all remote subscribers who have subscribed to sources local to this host.

The user can use a single listener object as the sink for more than one subscription. When this method is used it is up to the user to check and act on each different event type inside the listener object processEvent method.

Event Delivery and migration

The delivery of events to an event listener is located in the CProcessUserObject and not the event manager. This allows a segment to be migrated and for its CProcessUserObject to pass all events delivered during migration on to its new location. The

event delivery mechanism is implemented in the class eventDispatcher.

Vectors of a segments current subscriptions and sources are also maintained in CSegment. This information is used internally during migration to UN-Register all sources and UN-Subscribe all subscriptions held by this segment. This is done after the new moved version of the segment has used these same vectors to re-register all its current sources and re-subscribe all its current subscriptions. In the case of a new segment these vectors will be empty.

Events left in the delivery buffer after the source segment has UN-subscribed are packed into a vector and moved to the new segment where they are delivered to the listeners prior to the new vectors dispatcher thread delivering any events that have accumulated in its new dispatch queue.

6.5 Load Balancing

All load balancing functions are located in the class CLoadManager. An instance of CLoadManager is created and hosted in Nodesvr. Each time a new segment is loaded onto this server the load is incremented, and when a segment finishes or moves the load is decremented. CLoadManager runs a Thread that periodically checks the load status of the server. If the load exceeds a hi-water mark, loaded in by the policy manager then the Load manager will attempt to move a segment to the lowest loaded machine on the virtual network. The method for finding the destination machine is located in the getNewHost method. This method is also used by the migration service to locate a suitable host for migrating segments to.

6.6 The GUI

The GUI is implemented using the Java AWT. The class responsible for displaying and controlling the GUI is NodesvrGUI. Figure 6.5 shows a screen shot of the servers GUI. Much of the code used to implement the NodesvrGUI class was modified from examples found in [9]

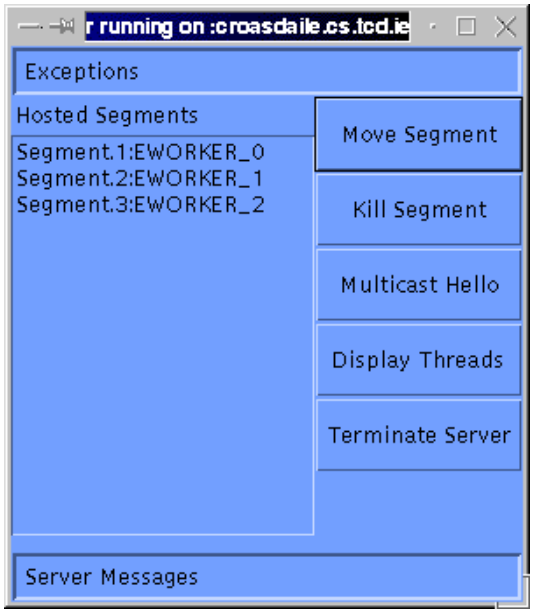


Figure 6.5: The server GUI

6.7 The Worm

The worm in this system is made up of a set of 1 or more segments bound together by a common process id. A user creates a worm by instantiating the class CCreateWorm, The constructor takes a single parameter, the number of segments the new worm will have.

The user code is added to each segment by instantiating an object of the user class for the segments, and calling the sendSegment method, passing in as its only parameter a reference to the newly created user object. Originally the worm had been designed to take only homogeneous segments. This restriction was later lifted but now has consequences when a dead segment is being replaced. Once all the segments have been received then CCreateWorm will contact the server on the same machine and ask that it host segment 0. All other segments will be dispersed across the virtual network, to maximize fault tolerance and computing resources. Figure 6.6 shows the code required to create and disperse a new worm. Note each segment can have a user defined name as well as the segment number assigned by CCreateWorm.

```
public class etest {
  public static void main(String[] args) {
    // This worm has ten segments
    int totsegs = 10;
    CMySegment worker = null;
    // Instantiate
    CCreateWorm newWorm = new CCreateWorm(totsegs);
    // Create all the segments
    for(int i=0;i<totsegs;i++) {
      worker = new CMySegment();
      worker.setName("WORKER_"+i);
      newWorm.sendSegment(worker);
    }
  }
}
```

Figure 6.6: The event Interface, IEvent

6.7.1 Recreating dead segments

One of the attributes of a worm is that when a segment dies, the worm will replace the dead segment automatically.

The first problem is, how is segment death defined. In this system death is the absence of communications for a user defined period. Each segment of the worm implements a heartbeat event. The heartbeat system is controlled by the CHeartBeat class. The CHeartBeat class has two functions. The first is to raise a heartbeat event at regular intervals. The second to monitor one other of the worms segments heartbeats. Monitoring is done as per the design using a ring structure.

One interesting change made to the event manager, was the introduction of a special event type for the heartbeat event. This type circumvented both the dispatch and receive queues and was delivered before all other waiting events. This was required due to the timing constrains governing heartbeats, as opposed to the asynchronous nature of events in general. With more time it would be useful to build a general purpose priority based event model into the event manager.

Dead segment are re-created by their active monitor. The active monitor

request the dead segments class file from its home server. This class is then used to instantiate a new copy of the dead segment. The server is requested to disperse the new segment. Dispersal is used to increase the fault tolerance of the segment. An active monitor and the segments that it monitors, should not, if possible, be hosted on the same server.

6.7.2 Life after death

Although the structures are in place this part of the system has not been implemented.

This chapter has discussed the implementation of a fully functioning worm and all its required services. Much of the development time was spent on implementing the event manager. Many concurrency issues had to be resolved, especially the scheduling of critical threads that handled network services. Migration using class loading and Java serialization appears straight forward, this is only true if you fully conversant with the concept of name spaces as implemented in Java.

Chapter 7

System usage and evaluation

This chapter covers the work done to evaluate the worm and the event system. Two applications were written that used both the worm paradigm and events as their framework. The successful implementation and running of the applications have in themselves shown that the concept of using events to implement a mobile worm will work.

In Chapter one we set out that the requirement of a worm was to provide the following four major services to its clients:

1. Transparent migration.
2. Location independent communications.
3. Segment fault tolerance.
4. Computational parallelism.

but we also wanted to provide the above services using the following constraints:

1. The worm should be fully distributed.
2. The code to implement the worm should have a small footprint.
3. The worm should be easy to use.
4. The worm should use an asynchronous and anonymous communications model.

5. The running worm should require the minimum amount of compute and communications resources.

6. The worm should function in a heterogeneous environment.

7.1 Achievements

- Although the migration service requires the user to structure their code in a special manner, and prohibits the migration of running threads, it does, once these criteria are met, provide a robust and transparent service at runtime.
- The Event Manager does provide a fully location independent and reasonably scalable communications service.
- Segments are within the limits set out in the design, fault tolerant.
- The worm does offer Computational Parallelism as shown in the matrix multiply application. All this has been done within the constraints set out above.
- A load balancing system was developed and worked as per specification.

The evaluations carried out involve testing that the system worked to the design specification. In the course of development, data has been collected on the performance of the event manager and the migration service. The chapter ends with a set of conclusions that have been drawn from the work.

7.2 Applications

Two applications were built on top of the worm. The first application was built to test all the algorithms were working, as per the design of the system. The second application, Matrix multiplication was built to test the validity of using a real world application that could exploit its natural parallelism using the worm.

7.2.1 A simple event exchange worm

The first application designed to be hosted by the worm was very simple. Its purpose in life was twofold. First it was designed to test the worms ability to replace dead segments. Second it is used to test the event service and especially the ability of the event service to cope with migration.

The application, built using the CMySegment class, like all applications designed to be hosted by the worm, supports the ISegment interface. This means the applications has three main functions;

1. *init* This function is called once by the worm when the applications segment is first called. In this instance the init function is used to subscribe the segment to a finishedCalc event, it also registers itself as a source for the finishedCalc event.

2. *doWork* This function is very simple, it does a small calculation the raises a new finishedCalc event. The doWork is repeatedly called until an internal counter reaches a user defined cutoff point. The calculation updates a number of internal counters. These counters should remain intact after a forced migration. This can be used to test the migration code built into the worm.

3. *terminate* This function is called by the worm when the segment it about to terminate. In this instance the application uses this function to unsubscribe and unregister all its event handlers.

The application as it runs logs all its critical data to a disk file, which can be analyzed after the test run has been completed. A short extract from the segment log file is shown in figure 7.1

The log proved especially useful in confirming the correct operation of the event service. It was useful for checking that events were not lost during segment migration. To show this one can take both the segment log on the source host and the segment log of the same segment from the destination host and ensure that all events have

```
Called function init
Event from :EWORKER_1 to EWORKER_2 Seq No.:2 from host :sun21
Event from :EWORKER_2 to EWORKER_2 Seq No.:1 from host :sun27
Event from :EWORKER_0 to EWORKER_2 Seq No.:4 from host :sun20
Event from :EWORKER_0 to EWORKER_2 Seq No.:5 from host :sun20
Event from :EWORKER_1 to EWORKER_2 Seq No.:3 from host :sun21
Event from :EWORKER_2 to EWORKER_2 Seq No.:2 from host :sun27
Event from :EWORKER_0 to EWORKER_2 Seq No.:6 from host :sun20
Event from :EWORKER_1 to EWORKER_2 Seq No.:4 from host :sun21
Event from :EWORKER_0 to EWORKER_2 Seq No.:7 from host :sun20
Event from :EWORKER_2 to EWORKER_2 Seq No.:3 from host :sun27
Event from :EWORKER_1 to EWORKER_2 Seq No.:5 from host :sun21
Event from :EWORKER_2 to EWORKER_2 Seq No.:4 from host :sun27
Event from :EWORKER_0 to EWORKER_2 Seq No.:8 from host :sun20
Event from :EWORKER_1 to EWORKER_2 Seq No.:6 from host :sun21
```

Figure 7.1: The segment log file contents

be delivered. Note all events have a sequentially increasing event number, therefor checking correct delivery is a straight forward process.

During a run segments were periodically killed to check that the worm behaved correctly. In this application, and restricting the worm design to cater only for homogeneous segments, the worm would survive and re-build itself as long as just one segment was alive at any point in time.

Correct termination behavior of segments was also tested by setting the termination counter in the application to a reasonable number. When a segment terminates it should unsubscribe and unregister all its sources. This was tested and found to operate to specification. The behavior of the heartbeat monitor was also tested for correct implementation when one segment finished before the others. A server was run on Wilde, which is considerably faster than the other test machines. When a segment terminates the heartbeat active monitor should be transfered to monitoring what the terminated segment was monitoring. In effect this works like a circular linked list when a single node is removed. This worked as per specification.

7.2.2 Matrix multiply

The Matrix multiply application was written to test a substantial real world application that can exploit parallelism. The application is comprised of two separate type of segments, segments are therefor not homogeneous. This fact has consequences for the worms design as will be shown later.

There is a single master segment, whose job is to allocate work, keep track of outstanding allocations, collate completed work, and when all work is completed to assemble the result matrix and write the output to a file.

The master segment is implemented using the CMasterMatrix class. The master allocates work at the request of workers. When it receives a requestWork event, it checks its work allocation table. If there is no work left to be allocated, it ignores the requestWork event, otherwise it updates an unused work allocation record, and generates a new workAllocation event setting the dest field in the event to the segment name that requested work. It also enters a start time into the allocation entry. If the master does not receive a reply, in the form of a finishedRound event, within a defined period, it will re-allocate the work to a new worker. If two segments eventually return the same work, the second result is ignored because the allocation entry has been marked as completed. When all allocation entries have been marked as completed, the master will output the completed matrix to file. Its final job is to send out a terminateAll event that will kill off itself and all the workers.

The workers are much simpler organisms,and is implemented using the CSlaveMatrix class. When a worker start it immediately sends out a requestWork event. If the worker does not receive a workAllocation event, it will continue to send out requestWork events once every eight seconds. When the worker does eventually receive a workAllocation event it does a matrix multiply and returns the answer, using a finishedRound event, all within one doWork cycle. Workers will terminate on reception of a terminateAll event.

Because all the segments in the worm are not homogeneous, two caveats apply to the fault tolerant behavior. One only one segment can fail at any point in time,

and two the home server where all the class files reside must not fail.

This application as we have seen is a much more substantial application and uses many different type of user defined events. It can be reported that the application worked correctly.

One problem arose when the matrix size was increased to 600 by 600. The application would mysteriously stop for up to forty seconds at a time. This lead to the monitors recreating segments which were not dead. This delay was eventually traced to the Java garbage collector. If you run Java with the verbosegc switch it will trace the garbage collector output to the console. This problem was alleviated by making explicit calls to the garbage collector directly within the server at convenient times.

7.3 Evaluation

As we have already stated both applications worked, and as such proved the concept. This by no means insures that the worm, as implemented here is a valid option for distributed computing. Much more testing would be required to guarantee robust behavior. Two other criteria must also be considered;

1. Is the system Efficient.

2. Is the system Transparent.

To answer the first point, No, at present the system is not efficient. It can take between 800ms and 8 seconds to move segments, depending on their complexity, and the load on the server. Many of the issues of inconsistent timings appear to be the result of two factors, concurrency and garbage collection. It is very difficult to be sure the implementation of the concurrency model is functioning as per the design specification. Without proper tools that can trace the life cycle of threads, and report on threads that appear to be waiting an inordinately long time to acquire object locks it is very difficult to be certain the model has been implemented correctly. Garbage collection is a problem when the server is busy. Hints can be passed to the garbage

collector to do its collections at reasonable times. When is a reasonable time in a system that uses asynchronous communications?

At present as each segment is unserialized, Java looks for the class files of the objects that make up the segment recursively. This leads to a a great deal of small network transfers, until all class files for objects referenced by the segment have been migrated. Overcoming this problem may be possible by moving all the class files required by the segment, prior to serializing the segment itself. This is possible in the system because all class files for a segment are already held in the class loader on the source host and are therefor known prior to the segment move.

Event delivery times can also be unpredictable, for a 300 byte event the average transport time is 50ms. Not being happy with this, a simple server was developed to test the time taken to move events between two machines, without any other load on the system. The average times for event delivery are shown in figure 7.2.

Size in bytes	Time in Milliseconds
1	50
100	50
1000	52
10,000	16
1,000,000	1048

Figure 7.2: Event Timings versus size

The data in figure 7.2 may seem strange, in particular the fact that moving an an event with a data packet of one byte takes 50Ms whereas an event with a packet size of 10,000 bytes takes 16Ms. It was suspected that this may be due to Nagle's algorithm [46]. Unfortunately when Nagle's algorithm was switched off using the setTcpNoDelay method from the socket class, stream errors were encountered in the Java serialization code.

The servers were tested under Java 1.2 using both the green and native threading model on Sun workstations. Under the native thread model the GUI interface was more responsive. There did not appear to be any other significant differences.

7.3.1 System Constraints

All systems have constraints, but it is desirable to minimize these constraints to allow users of the system to gain the most benefit from that system. Overall the largest constraint in the system is forcing the user to implement their code in a predetermined manner. Their segments must inherit from CSegment and must implement the ISegment interface.

Catching errors at compile time or run time

Imposing the constraint that all user classes must implement the Java serialization interface, cannot be tested at compile time. The system has no means of forcing this constraint at compile time and if, as is probable, the user omits implementing this interface, Exceptions will be thrown in the migration service. The exceptions themselves will not harm the server but the migration service will refuse to migrate the offending segment, these types of errors could lead to a great deal of confusion for the user.

Network partitions

If there is a network of 30 servers running, and for some reason the network gets partitioned, with 12 servers active in one partition and 18 servers active in the other partition, then if there is a worm that has segments hosted in both partitions, there will be problems. This leads under the present design, to the recreation of two identical worms, one on each partition, if the failure is corrected and the partitions merge two identical worms will be on the virtual network at the same time. The system will not cater for this type of failure. ISIS [5] has an algorithm that overcomes this problem.

Running Threads

The user cannot have there own threads running when the segment returns from a call to doWork. This like the previous constraint cannot be detected at compile time,

but must be caught by the use of runtime exceptions.

Heterogeneous segment restrictions

There are two flaws with the design for re-creating dead segments that future work should address. First if the monitored segment dies before sending any heartbeat, the monitor cannot re-create it, because it has no access to its class file. This short coming can be overcome by putting this information directly into all segments prior to dispersal. The second problem occurs if the home server goes down, then the class file cannot be retrieved. The solution to this is to duplicate all the class files for a worm and have two home nodes, the main one acting as the primary class resource and the duplicate as the secondary or backup.

When a segment has state independent of other segments

The worm at present does not save any state, so when a segment dies all knowledge of its state dies with it. Setanta Mathews [34] developed a worm that used hot shadows to save a segments state. This could also be used in this system.

When a segment has state dependent on other segments

The current system has not been designed to work under these conditions. To do so requires event ordering and distributed check-pointing. If a system of this nature is required then group communications products such as ISIS, or for Java, iBus [22] would be a better choice.

The Event Model

The event model works well, and due to its distributed nature should scale reasonably. At present it does not implement constraints. This omission became evident when using the Matrix multiply application. The work allocation events contain parts of large matrices and even though each event was only destined for one of the workers, a copy was sent to all workers and filtering only occurred at the destination. Filtering at source as shown by [19] can optimize network usage.

7.4 Conclusions

This dissertation has presented the concept of using the events paradigm to implement a worm in a mobile environment. Using the development of a worm with both migration and event services, has shown the concept does work.

A number of useful lessons have been learned as a result of the implementation of the project, to summarize briefly they are as follows;

- The overhead of serializing and moving objects in Java imposes an upper bound on system performance.
- Debugging distributed systems is difficult, and there are few tools to assist the developer.
- Achieving real transparency in a distributed system comes with a high price tag for the systems developer.
- The more centralized a system is the easier it is to design and develop, but the less reliable in the face of failure. The best solutions usually lead to a compromise between the amount of development time available and the need for fault tolerance.

Much has been achieved in the development of this dissertation, but there are a number of issues that could be looked at in the future. First and foremost would be to make migration fully transparent. To achieve this goal may well require work down at the Java Virtual Machine level [16].

Another future enhancement would be to improve the handling of segment death, in particular the ability to checkpoint a worm and allow rollback in case of failure, rather than re-creating segments and loosing all their state. Hot shadows could be used for this purpose and this was the direction taken in [34].

The system as it stands is capable of behaving as an agent hosting environment, it would therefor be useful to allow segments to request their own migration and also to allow them to migrate to preferred servers.

The event model should be extended to include user definable constraints to be evaluated at source. At present the design optimizes the sending of events from a source to a set of subscribers on one host by only sending one event, the event manager clones and distributes that event to all subscribers on its host. If constraints at source are implemented, it would be interesting to look at the possibility of implementing an algorithm for coalescing all the constraints from subscribers on one host into a single constraint on the destination which would still keep the transport of events across the network to a minimum, and the evaluation of constraints at source efficient.

In conclusion this dissertation has researched, designed and implemented a worm that works in a mobile environment. In developing this worm, a robust migration service was built and a fully distributed event service was implemented. This service is capable of transparently handling the migration of both its sources and sinks. Fault tolerance has been built into the system to handle the premature death of worm segments and finally a load balancing system for the virtual network was implemented.

Bibliography

[1] Jim Waldo Ann Wollrath and Roger Riggs. Java-centric distributed computing. *IEEE Micro*, 17(3), 1997.

[2] Ken Arnold and James Gosling. *The Java Programming Language Second Edition.* Addison Wesley, 1998.

[3] Steve Benford and Ok-Ki Lee. Collaborative naming in distributed systems. *Distributed Systems Engineering*, 7(2), 1993.

[4] Kenneth P. Birman and Thomas A. Joseph. Reliable communication in the presence of failure. *ACM Transactions on Computer Systems*, 5(1), 1987.

[5] Kennith P. Birman and Robert Van Renesse. *Reliable Distributed Computing with the ISIS Toolkit.* IEEE Computer Society press, 1994.

[6] A. D. Birrel and R. M. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2, 1984.

[7] J. Brunner. *The Shockwave Rider.* Dent, 1975.

[8] Nicholas Carriero and David Gelernter. Linda in context. *Communications of the ACM*, 32(4), 1989.

[9] Patrick Chan and Rosanna Lee. *The Java Class Libraries Second Edition, Volume 2.* Addison Wesley, 1998.

[10] D. R. Cheriton. The v kernel: A software base for distributed systems. *IEEE Software*, 1(2), 1984.

[11] Douglas E. Comer. *Internetworking with TCP/IP Volume 1.* Prentice Hall, 1991.

[12] The Unicode consortium. *The Unicode Standard.* Addison-Wesley, 1996.

[13] Frederick Douglis Dejan Milojicic and Richard Wheeler. *Mobility processes, computers and agents.* ACM Press, 1999.

[14] Dreyfus and Dreyfus. *Mind Over Machine.* Basil Blackwell Ltd., 1986.

[15] Hupfer Freeman and Arnold. *JavaSpaces Principles, Patterns and Practice.* Addison-Wesley, 1999.

[16] Stefen Funfroken. Transparent migration of java-based mobile agents. Technical report, Darmstadt University of Technology, Dept. Computer Science, 1999.

[17] Object Management Group. *The Common Object Request Broker: Architecture and Specification, 2nd ed.* Object Management Group, 1995.

[18] Dominique Steve Guy Bernard and Michel Simatic. A survey of load sharing in networks of workstations. *Distributed Systems Engineering*, 7(1), 1993.

[19] Mads Haahr. Implementation and evaluation of scalability techniques in the eco model. Master's thesis, University of Copenhagen, Dept. of Computer Science, 1998.

[20] Mark Garland Hayden. *The ENSEMBLE System.* PhD thesis, Cornell University, 1998.

[21] Michi Henning and Steve Vinoski. *Advanced CORBA Programming with C++.* Addison-Wesley, 1999.

[22] SoftWired Inc. Developing publish/subscribe applications with ibus. Technical report, SoftWired Inc., 1999.

[23] Bill Joy James Gosling and Guy Steele. *The Java Language Specification.* Addison Wesley, 1996.

[24] Richard Hayton Jean Bacon, John Bates and Ken Moody. Using events to build distributed applications. In *Proc IEEE SDNE Services in Distributed and Networked Environments*, 1995.

[25] Richard Koo and Sam Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Transactions on Software Engineering*, 13(1), 1987.

[26] L. Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7), 1978.

[27] Danny B. Lange and Mitsuru Oshima. *Programming and Deploying Java Mobile Agents with Aglets.* Addison-Wesley, 1998.

[28] Doug Lea. *Concurrent Programming in Java.* Addison-Wesley, 1997.

[29] Patrick Chan Rosanna Lee and Douglas Kramer. *The Java Class Libraries Second Edition, Volume 1.* Addison Wesley, 1998.

[30] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification Second Edition.* Addison Wesley, 1999.

[31] Litzkow M. and Solomon M. Supporting checkpointing and process migration outside the unix kernel. In *Proceedings of the USENIX Winter Conference*, 1992.

[32] Powell M. and Miller B. Process migration in demos/mp. In *Proceedings of the 9th ACM Symposium on Operating Systems Principles*, 1983.

[33] Keith A. Lantz Marvin M. Theimer and David R. Cheriton. Preemptable remote execution facilities for the v-system. *ACM Operating Systems Review*, 119(5), 1985.

[34] Setanta Mathews. Building reliable distributed applications using the worm paradigm. University of Dublin, 1999. Dept. of Computer Science Final year project.

[35] Sape Mullender. *Distributed Systems.* Addison-Wesley, 2nd edition, 1993.

[36] Scott Oaks and Henry Wong. *Java Threads Second Edition.* O'Reilly, 1999.

[37] Karl O'Connell. *System Support for Distributed Multi-User Virtual Worlds.* PhD thesis, University of Dublin, 1997.

[38] O'Connor, M., B. Tangney, and V. Cahill. Harris, N. micro-kernal support for migration. (also to appear in distributed systems engineering journal) . Technical Report TCD-CS-94-05, Department of Computer Science, University of Dublin, Trinity College, 31 August 1994. Fri, 18 Apr 1997 12:38:03 GMT.

[39] Kenneth P. Birman Robbert van Renesse and Silvano Maffeis. Horus: A flexible group communications system. *Communications of the ACM*, 39(4), 1996.

[40] Donn Seely. The internet worm of 1988. http://www.mmt.bme.hu/ kiss/docs/opsys/worm.html.

[41] Sheng and Gilad Bracha. Dynamic class loading in the java virtual machine. *ACM OOPSLA 98*, 1998.

[42] John F. Shoch and Jon A. Hupp. The "worm" programs-early experience with a distributed computation. *ACM*, 25(3), 1982.

[43] David A. Solomon. *Inside Windows NT Second Edition.* Microsoft Press, 1998.

[44] Gradimir Starovic, Vinny Cahill, and Brendan Tangney. The ECO model: Events + constraints + objects. Technical Report TCD-CS-95-05, Department of Computer Science, Trinity College, 1995.

[45] Andrew S. Tanenbaum. *Distributed Operating Systems.* Prentice Hall, 1995.

[46] Andrew S. Tanenbaum. *Computer Networks.* Prentice Hall, 1996.

[47] Steve Vinoski. Corba: Integrating diverse applications within distributed heterogeneous environments. *IEEE Communications*, 11(3), 1997.

[48] Jim Waldo Geoff Wyant Ann Wollrath and Sam Kndall. A note on distributed computing. http://www.sun.com/research/techrep/1994/abstract-29.html.