Kami: An Anarchic Approach to Distributed Computing.

by

Jacky Mallett

B.Sc.(Hons.) Computer Studies. Loughborough University of Technology

Submitted to the Program in Media Arts and Sciences,
School of Architecture and Planning,
in partial fulfillment of the requirements for the degree of
Master of Science in Media Arts and Sciences

at the
Massachusetts Institute of Technology

September 2000

© Massachusetts Institute of Technology 2000. All rights reserved.
Kami: An Anarchic Approach to Distributed Computing.

by

Jacky Mallett

Submitted to the Program in Media Arts and Sciences,
School of Architecture and Planning,
on August 29 2000, in partial fulfillment of the
requirements for the degree of
Master of Science in Media Arts and Sciences

Abstract

This thesis presents a distributed computing system, Kami, which provides support for applications running in an environment of heterogeneous workstations linked together by a high speed network. It enables users to easily create distributed applications by providing a backbone infrastructure of localized daemons which operate in a peer-to-peer networking environment, providing support for software distribution, network communication, and data streaming suitable for use by coarse grained distributed applications.

As a collective entity, kami daemons, each individually run on a single machine, form a co-operating anarchy of processes. These support their applications using adaptive algorithms with no form of centralized control. Instead of attempting to provide a controlled environment, this thesis assumes a heterogeneous and uncontrolled environment, and presents a model for distributed computation that is completely decentralized and uses multicast communication between workstations to form an ecology of co-operating processes, which actively attempt to maintain an equilibrium between the demands of their users and the capabilities of the workstations on which they are running.

Thesis Supervisor: V. Michael Bove, Jr.
Title: Principal Research Scientist, MIT Media Laboratory
Kami: An Anarchic Approach to Distributed Computing.

by

Jacky Mallett

The following people served as readers for this thesis:

Thesis Reader

Andrew Lippman
Associate Director
MIT Media Laboratory

Thesis Reader

Oliver Selfridge
Senior Lecturer
MIT Media Laboratory
Acknowledgements

I would first of all like to thank Mike Bove, my supervisor, for allowing me to join his group, and for his encouragement and support whilst writing this thesis.

I would also like to thank Oliver Selfridge for a number of stimulating discussions on adaptiveness that significantly influenced the direction this work has taken.

In addition Charles Keen of Nortel Networks who was kind enough to provide the Accelar switch used for the backbone network communications for this project, and Ari Benbasat whose chosen method of thesis avoidance so greatly improved the latex presentation of this one.
Things fall apart,
the centre cannot hold.
Mere anarchy is loosed
upon the world.

W. B. Yeats.
Contents

Abstract 3

Contents 9

List of Figures 13

List of Tables 15

1 Introduction 17

2 Decentralized Computation 21
   2.1 Introduction ............................................. 21
   2.2 Clusters .................................................. 21
   2.3 Examples of Existing Systems ......................... 22
      2.3.1 Beowulf ............................................. 23
      2.3.2 Condor .............................................. 23
      2.3.3 CORBA .............................................. 24
   2.4 Distributed Programming Models and their network counterparts .... 26
      2.4.1 The Multicast Communication Protocol ............ 28
      2.4.2 A multicast model for distributed computation .... 31
      2.4.3 Dealing with unreliability .......................... 31
      2.4.4 Adaptation and distributed computing ............. 32

3 Design Considerations 35
   3.1 Local Responsibility for Global Integrity ................ 36
   3.2 Programming Model ....................................... 36
3.2.1 Kami Rings ........................................... 37
3.2.2 Messaging ........................................... 38
3.2.3 Source code distribution .............................. 39
3.3 Decision Making and Adaptiveness in a Kami Network .................. 40
  3.3.1 Self Modifying Code ................................ 41
3.4 Resource Scheduling and Allocation ............................... 42
  3.4.1 The Kami Marker Economy ........................... 42
3.5 System Control and Monitoring ................................ 43
  3.5.1 Source Code Handling ............................... 43
  3.5.2 Crash Monitoring and Handling ....................... 44
  3.5.3 User Applications .................................. 46
3.6 Message Implosion ....................................... 48

4 Kami Applications. ..................................... 51
  4.1 Considerations ......................................... 52
    4.1.1 Name Space Considerations ...................... 52
    4.1.2 Kami file space and directory structure ........ 52
4.2 Kami Rings ............................................. 54
  4.2.1 Ring support ....................................... 54

5 Writing a Kami Application .................................. 57
  5.1 Real Time Streaming: Quiz TV ............................. 57
    5.1.1 Application sub-systems. ......................... 58
    5.1.2 Modus Operandi .................................... 59
  5.2 Distributed Computation: Traffic Lights .................... 60
    5.2.1 Application subsystem .............................. 61
    5.2.2 Observations ....................................... 66

6 Future Directions .......................................... 69
  6.1 Support ............................................... 69
  6.2 Security ............................................... 69
  6.3 Wide Area Operations .................................... 70
  6.4 Scaling ................................................. 71
List of Figures

2-1  Centralized vs. Distributed Network Topologies .......................................... 26
3-1  Qtami Graphical Display of a five machine Kami Network ............................ 40
4-1  Kami directory organisation ............................................................................ 53
5-1  Quiz TV Distributed Program ........................................................................... 58
5-2  Traffic Light Simulator ...................................................................................... 61
5-3  Output from the Traffic Light Simulation as it converges to gridlock ............ 67
A-1  Kami Parameter Structure ................................................................................. 74
A-2  Kami Interface Function Source Code ............................................................... 75
A-3  Setup and Control Application perl script ....................................................... 76
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Summary of Distributed Systems</td>
<td>25</td>
</tr>
<tr>
<td>B.1</td>
<td>C and C++ support for sending Kami Functions and Messages</td>
<td>78</td>
</tr>
<tr>
<td>B.2</td>
<td>C and C++ utility functions</td>
<td>80</td>
</tr>
<tr>
<td>B.3</td>
<td>Kami System Commands</td>
<td>82</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Sometimes an application just needs more than one computer. Perhaps one isn’t powerful enough, or because a single computer can’t handle all the peripherals the application needs, or simply because elements of the task are geographically separated. When this is so, distributing the application over several computers should not be a difficult task, and yet in practice it often is. Relatively little support is currently provided for these applications beyond basic networking capabilities.

When ancient cartographers offered the advice 'Here be Dragons' on the lesser traveled areas of their maps, it warned the timid to stay at home, and the more foolhardy that they should at least exercise some care when venturing. Were computer scientists to publish maps, endeavors in distributed computing might well be subject to a similar label, an area known to exist, to have been explored, and even suspected to have some inhabitants, but still one which potential visitors should think carefully about before traveling into. Putting together a distributed application in the current computational environment is something that requires careful consideration and a lot of work.

Ironically, as today’s users sit in front of their networked workstations, they are using an advanced and highly reliable form of decentralized computation, the network switches themselves. They function routinely as autonomous computation units of some considerable complexity. Their users who wish to create an advanced and highly reliable distributed application need to be unfortunately well versed in the complexities of data communications;
especially if the application does not fit easily into the standard client-server model of distributed computation.

There are good reasons why most applications remain confined to a single machine, and twenty year old distributed operating systems such as Locus[17] are not found on every desktop. There may not be dragons in distributed computing, but there are a great many hard problems and an often prohibitive increase in complexity and overhead involved in providing satisfactory solutions to them. As Popek[17] points out, distributed systems have a much richer set of error modes than those of a single system. They also have to deal with network issues of latency and reliability and, until recently, limited bandwidth availability.

With the large increase in network speeds arising from the introduction of fibre-optic data links, the attraction of distributing computation over several computers becomes more and more compelling. And as computers become cheaper there are more and more of them available to run such applications. An example of the influence of increasing communication speeds and decreasing component costs can already be seen in the hardware of individual computers. These have evolved from a single cpu into a set of highly specialized computational elements, each handling a relatively self contained task, such as video or audio processing. Now that network speeds are comparable to the backplane computer speeds of low end workstations, the hardware environment for distributed applications is set for a similar evolution.

On examining today's distributed applications, several things are apparent. Almost all distributed applications use the client-server model for their architecture. The underlying TCP/IP protocol tends to impose this model, since it is a point-to-point protocol and requires a pre-defined sender-receiver asymmetry. Consequently, distributed applications tend to be highly centralized, and support little or no direct client to client communication. This is acceptable for systems that are intrinsically centralized, such as database applications, but it is problematic for those that are not. And as noted, it is based on a network protocol that was designed with a set of assumptions in mind that in high speed network environments is no longer strictly applicable.

Although even centralized applications may have issues with the single point of failure represented by the server(s).
So assuming that the network connecting a set of workstations is considerably faster than all other potentially computation blocking elements in the system, what sort of support can be usefully supplied for distributed applications in such an environment? In effect, the historical relationship between host computer and the network has been reversed. The bottlenecks for communication are becoming the sources and sinks of data, rather than the network between them. The amelioration of the network problem however, does not in and of itself offer a panacea to the problem of distributed applications. Hardware constraints are not the only barriers to the effective construction of distributed systems.

As Pfister[16] points out, the real bottleneck in any distributed system does not typically lie in any particular hardware arrangement, but in creating the software that runs on that system. Any non-trivial programming is a challenge in the homogeneous and controlled world of the single computer. Using several machines involves the far greater challenge of imposing order and predictability onto an intrinsically heterogeneous networked environment with a far richer set of error conditions.

Add to this Amdahl's observation[2] on the diminishing returns from parallelism, essentially that the speed of any parallel calculation will always be limited by the speed of the sequential elements in the calculation. Combined with the steady increase in cpu speeds popularly referred to as Moore's law, has caused many of the early efforts in parallel hardware to be outperformed by sequential computers within a remarkably short time[4]. These factors have posed a formidable obstacle to efforts to construct bespoke distributed hardware solutions.²

The goal of this thesis is to provide software support that enables users to readily distribute applications over a set of networked workstations, without having to worry about underlying network handling or machine availability. This thesis aims to facilitate their implementation regardless of whether the application is a sophisticated weather simulation with extreme computational requirements, a set of intelligent remote sensors that need to share data, or a video processing program that simply needs more video cards than are typically found on a single machine.

To achieve this a distributed programming daemon has been created called kami, that runs

²Hank Dietz has an interesting summary of hardware based parallel efforts titled "Is Parallel Processing dead?" at http://dynamo.ecn.purdue.edu/~hankd/Opinions/pardead.html
autonomously on each participating machine. It can compile load and run locally supplied source code, provides applications with a simple but robust interface to the networking communication layer, and can also accept input or control commands from direct TCP/IP connections to the daemon.

The system is anarchic, in the sense that there is no centralized control of any form nor any requirement, beyond support for the common messaging interface, for daemons to follow the same set of rules when making decisions. This distinguishes it from several similar systems, which often have at least an element of centralized control and administration. It also differs in providing automatic support for user source code distribution, and in providing internal monitoring of user code performance to adaptively maintain local system stability.

Some of the problems associated with distributed computing are formally unsolvable, some are simply very complex and require more programming effort than is generally economically feasible, and others arise from limitations in the network protocols and operating system support currently used for computer communications. This thesis does not tackle any of these hard problems, it simply aims to help applications that want to use more than one computer to do something useful.

---

3 The complete Oxford English Dictionary provides five definitions of anarchy, the one used here is the second: "A theoretical social state in which there is no governing person or body of persons, but each individual has absolute liberty (without implication of disorder)."
Chapter 2

Decentralized Computation

2.1 Introduction

Performing applications on multiple computers has a long history, and a correspondingly varied nomenclature. Parallel programming, the execution of identical instructions on different computational elements, lies at one end of a continuum that stretches to distributed real-time computation, which can be broadly described as the execution of quite different instructions on computers that are collaborating at some level of abstraction on a shared task.

In the world of distributed computation a further distinction can also be made between systems whose application topology is in some way centralized, as in the client-server model and those which are completely distributed, sometimes referred to as decentralized computation. At the workstation level, the term cluster is often used to refer to a set of co-operating machines, although this term is also subject to some discussion about its exact definition.

2.2 Clusters

Clusters are defined by Pfister[16](p72), as a "collection of interconnected whole computers that can be used as a single unified computing resource". Under Flynn’s[9] accepted tax-
onomy of distributed computation models they fill the Multiple Instruction Multiple Data, and the supposedly mythical, Multiple Instruction, Single Data organizational niches.\(^1\)

Although Pfister\(^\text{(16)}\)(p300), and others, have downplayed the existence of the Multiple Instruction Single Data (MISD), form of parallel computing; this may have owed more to slow networking speeds than any real infeasibility of that model. For example, processing video streams to simultaneously find different types of objects can be usefully done by several different algorithms running simultaneously on independent machines. There are several instances where a large data set requires multiple independent and time consuming operations to be applied: for example, trading market simulations and analysis, data mining problems as a class, and some genetic algorithm implementations. Given sufficiently high data speeds between machines this kind of problem can be easily and profitably distributed in an MISD environment.

Describing clusters as providing a single unified computing resource leaves considerable scope for a wide degree of interpretation on what constitutes a unified computing resource. Existing systems range from the linux based Beowulf\(^\text{(18)}\), which explicitly take over the whole computers and present the user with the exact semblance of one system, accessed through a single node, to systems such as Condor from the University of Wisconsin\(^\text{(11)}\), which provides a batch processing environment within a network of desktop workstations, and which actively moves jobs around the network to unused machines, switching between computers when user activity intervenes.

Somewhere in between these extremes lies the requirements of a user who wishes to put together a simple application that uses a set of networked machines with heterogenous capabilities and resources.

### 2.3 Examples of Existing Systems

Once the workstation established itself as a common desktop tool, systems for using workstation clusters in various different parallel computing models began to proliferate. Most

\(^1\)Flynn's taxonomy also describes the typical desktop computer, as the Single Instruction Single Data model, and includes the Single Instruction Multiple Data model which describes the most widely used form of parallel computation, data pipelining.
of these systems were designed to tackle heavy computational problems, rather than the distributed use of a more generalized concept of computer resources.

Overviews of some of the most common systems in use today are given below; table 2.1 shows a summary of other currently available systems, drawing on Kaplan's [10] report which provided an overview of the situation in the mid-1990's, Pfister[16], and Tanenbaum[20].

2.3.1 Beowulf

Beowulf originated from a project sponsored by the High Performance Computing and Communications program[18]. Its primary goal is to provide cheap cluster computing for problems with large computational requirements. Beowulf systems usually consist of computers dedicated solely to running Beowulf, with access to the system via one centralized node.

Beowulf is an example of a system that can be used to construct a parallel computer from a set of dedicated workstations, rather than a system that co-exists with other users and attempts to steal computer cycles. Its advantages are that it uses a modified version of the linux operating system, and can operate within a highly heterogeneous set of computers. It is particularly useful for building processor farms from obsolete, low power computers.

It should also be noted that a number of installations have progressed to Beowulf-style installations, which apply the Beowulf principle, but use linux utilities such as PVM and MPI to build their own infrastructure.

2.3.2 Condor

Condor is a system that provides a virtual mainframe environment for batch computing within a network of user workstations [11]. Condor provides a batch job submission environment that runs in the background of workstations participating in the condor group, and that actively attempts to steal idle cycles on unused workstations. Jobs are moved from workstation to workstation as real-time user activity changes.
Condor uses job checkpointing to enable it to move processes around the network, this makes it particularly useful for tasks that have extremely long computational requirement, order(days), since it offers protection against individual workstation crashes causing the loss of large amounts of computational work.

2.3.3 CORBA

The Common Object Request Broker Architecture (CORBA) is an open distributed object computing infrastructure being standardized by the Object Management Group, a large commercial consortium numbering several hundred. Its objective is to provide an environment within which distributed applications in an heterogeneous environment can be integrated. Network communications equipment providers in particular, are interested in using it as a means of integrating the large numbers of legacy systems that typify today’s network management environment.

As described in Vinoski[22] CORBA is a mixture of supporting software, primarily the CORBA Object Request Broker (CORBA ORB), which provides the networking support, and the Interface Definition Language which is used to construct CORBA interfaces for objects operating within the architecture.

As might be expected from its parentage and ambitions, in its various implementations and standards CORBA has become a complex, and multi-faceted architecture, with both advocates and critics; and it goes beyond the scope of this thesis to do justice to both sides of a debate, the results of which are yet to become apparent. CORBA does provide a standardized, general purpose interface as advertised, but it does so at the price of considerable complexity and overhead that makes it unsuitable for casual applications. It also provides no refuge from the client-server model, the assumption of which is built into its model, reflecting an intentional predisposition towards database access and integration in its design.
Table 2.1: Summary of Distributed Systems

<table>
<thead>
<tr>
<th>System Name</th>
<th>Type</th>
<th>Operating System</th>
<th>Requires NFS</th>
<th>Central Control</th>
<th>Source</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amoeba</td>
<td>o/s</td>
<td>Posix</td>
<td>No</td>
<td>Yes</td>
<td>Vrije Universiteit</td>
<td>Uses dedicated processors</td>
</tr>
<tr>
<td>Chorus</td>
<td>o/s</td>
<td>Chorus</td>
<td>No</td>
<td>Yes</td>
<td>Chorus Systems</td>
<td>Now branching out into micro-kernel and CORBA support</td>
</tr>
<tr>
<td>Codine</td>
<td>Batch</td>
<td>Unix</td>
<td>No</td>
<td>Yes</td>
<td>GENIAS Software GmbH.</td>
<td>Merger of DQS and condor</td>
</tr>
<tr>
<td>Condor</td>
<td>Batch</td>
<td>Unix</td>
<td>No</td>
<td>Yes</td>
<td>University of Wisconsin</td>
<td></td>
</tr>
<tr>
<td>CONNECT:Queue</td>
<td>Batch</td>
<td>Unix</td>
<td>Yes</td>
<td>Yes</td>
<td>Sterling Software</td>
<td></td>
</tr>
<tr>
<td>DQS</td>
<td>Batch</td>
<td>Unix</td>
<td>Yes</td>
<td>Yes</td>
<td>Florida State University</td>
<td>Commercialized as Codine</td>
</tr>
<tr>
<td>Load Balancer</td>
<td>Batch</td>
<td>Unix</td>
<td>Yes</td>
<td>Yes</td>
<td>Unison-Tymlabs</td>
<td></td>
</tr>
<tr>
<td>LoadLeveler</td>
<td>Batch</td>
<td>Unix</td>
<td>Yes</td>
<td>Yes</td>
<td>IBM</td>
<td>Modified commercial version of Condor</td>
</tr>
<tr>
<td>Load Sharing Facility</td>
<td>Batch</td>
<td>Unix</td>
<td>Yes</td>
<td>Yes</td>
<td>Platform Computing</td>
<td>Designed to handle very large clusters</td>
</tr>
<tr>
<td>Mach</td>
<td>o/s</td>
<td>Unix</td>
<td>No</td>
<td>Yes</td>
<td>Carnegie-Mellon University</td>
<td></td>
</tr>
<tr>
<td>NC Toolset</td>
<td>Batch</td>
<td>Unix</td>
<td>No</td>
<td>Yes</td>
<td>Cummings Group</td>
<td></td>
</tr>
<tr>
<td>Network Queueing</td>
<td>Batch</td>
<td>Cray,Solaris</td>
<td>No</td>
<td>Yes</td>
<td>CRAY Research Inc.</td>
<td></td>
</tr>
<tr>
<td>Portable Batch System</td>
<td>Batch</td>
<td>Cray,Unix</td>
<td>No</td>
<td>Yes</td>
<td>NASA Ames Research Center</td>
<td></td>
</tr>
<tr>
<td>Task Broker</td>
<td>Batch</td>
<td>Unix</td>
<td>No</td>
<td>Yes</td>
<td>Hewlett Packard</td>
<td>Uses bidding mechanism to distribute work</td>
</tr>
</tbody>
</table>
2.4 Distributed Programming Models and their network counterparts

As Pfister[16] points out, although the ideals of distributed and parallel computation are well understood, the many different implementations and their rationales are not as well defined. The most commonly used model for a networked application is the client-server model, which in networking terms uses a centralized star topology of communication between local clients, and a centralized server. This maps directly onto TCP/IP point to point communication protocols, as shown in Figure 2-1.

![Star Network (Client/Server Topology) vs. Completely Connected Network (Peer-To-Peer Topology)](image)

Figure 2-1: Centralized vs. Distributed Network Topologies

Another way to look at the client-server model of distributed computing is as a specialized implementation of a mainframe environment designed for a particular problem. In this model clients handle local processing of varying degrees of sophistication, and connect to servers which provide centralized control, co-ordination and access to centralized data resources as required by the application. Clients can range widely in their complexity, but the
key concept is that of a centralized point through which some element of system control or shared data must pass. Although servers can be duplicated and are often set up in their own clusters to share load and reduce system vulnerability, this model is inherently hierarchical in its organization.

A large class of problems falls naturally into the client-server model, especially ones that require centralized database access. This model becomes problematic when the distinction between client and server is not clear cut, particularly when the application seems to require an entity that combines the recognized characteristics of both client and server. Especially problematic are applications where the natural flow of communication within the application includes client to client communication, and those where the clients are not homogeneous, or where more than one type of server is required.

As an example, consider the fully computerized house of the putative future, where every domestic appliance is a fully networked device with its own micro-computer. How does one control such a thing? Under the client-server model, there would be a centralized computer, the server, which co-ordinates all activities in the house. The user would program a set of requirements, and the server would instantiate them, for example, alarm clock triggered at a particular time, coffee maker switched on five minutes before, lights switched on five minutes afterwards, etc.

The problem with this approach is that the server becomes extremely complex. A secondary problem, is the lack of local knowledge. For example, if the coffee maker doesn’t have any water, will the server still switch it on? If there is no motion detected in the house, should the heating still be set to the same temperature? Each of these problems can be resolved, the server checks the water level, for example; but at the price of adding extra complexity to the server. Complexity and software reliability tend to be mutually exclusive, especially in the context of distributed systems.

The alternative is to increase the capabilities of the local clients so that they can negotiate with each other, in other words to turn them into intelligent objects. Each one can query the other, the coffee maker for example can ask the alarm clock or the motion sensor for the information it needs, and then determine what is the most appropriate action to take locally. At the price of slightly more complicated local objects, it can be argued that the
overall system becomes more robust and flexible. With this model it is much easier to introduce new objects without disrupting service to the ones currently in place.

However, without changing the underlying communication topology, breaking away from the client-server model is difficult. To do so, a different network communication layer is needed. Ideally one that treats all participants as peers, with as little additional overhead as possible. The protocol that has been chosen for this purpose is multicast[5], which is a broadcast protocol with some useful addressing aspects.

2.4.1 The Multicast Communication Protocol

Multicast is a UDP based communications protocol available as part of the unix socket based TCP/IP communication stack. The UDP class of protocols is defined as unreliable, in that there is no sending and checking of packet acknowledgements to ensure data has been correctly received. The underlying communication layers will detect packet corruption, but no guarantees are made about packet receipt.

Multicast is an extension to simple host to host UDP communications, and offers a non-reliable communication interface which is one to many, rather than one to one. Delivery of a multicast datagram or packet is best effort only, and packets can arrive out of order, or be lost. There are also no guarantees that all hosts participating in a given multicast group will necessarily receive the same set of delivered packets. In practice however, unless the network is under heavy load, it is unusual for packets not to be delivered. Efforts are also underway to provide a reliable multicast protocol, Floyd [8] describes one approach, and RFC 2357[14] has been issued describing evaluation criteria. At this time, the consensus appears to be that due to the considerable technical issues involved, a single solution is unlikely, and a problem domain dependent approach is being favored.

It is also worth highlighting that now that the network is significantly faster than its host computers or data sources, it is no longer possible for the hosts to saturate the network equipment with traffic, even using broadcast protocols like multicast. As noted by Dittia [6] and others, today's hosts are not designed for high speed networks, and are incapable of coming anywhere close to achieving full band rate on a gigabit switch. This effectively
removes issues of network load from the arguments against using multicast, at least from the perspective of the network; hosts may still have local problems sending and receiving data if they cannot process their internal networking buffers fast enough.

For a distributed computing model, multicast has the immediate attraction that it is one to many, rather than one to one, which more naturally reflects the task in hand. Multicast communication is also somewhat simplified in comparison to TCP/IP, for hosts using multicast communication do not send messages to one specific host; rather they join a multicast group, and receive all communication sent to that group. Critically, hosts do not need to know which hosts are connected to the group, since this is delegated to the underlying data-communication equipment. Packets sent to a multicast group are automatically sent to all other members of the group, not necessarily including the packet originator, by the network switch or router.²

The IP address range 224.0.0.0 to 239.255.255.255 is allocated for multicast groups, and almost all of this range is available for use for dedicated multicast groups. There are no protocol limits on the number of such groups a particular host belongs to. Hosts join and leave multicast groups using the IGMP, (Internet Group Management Protocol, RFC 2236)[7]. Again in practice, there are hardware imposed limitations on the number that can be used by a single machine, but they have yet to be found restrictive.

From the perspective of a distributed computing architecture, multicast offers three advantages to direct host to host communication. Communications are considerably simplified and more accessible. There is no a priori need to know detailed addressing information about the participating hosts which simply need to agree to use the same multicast group address for their communication. Since there are no restrictions on the number of multicast groups a host belongs to, group addresses can be used to sub-divide the hosts into different communities of interest, and even direct host to host communication can be handled this way, by setting up multicast groups for each pair of hosts that requires it.

Against these advantages though, has to be placed the definition of multicast, which has

²Strictly it is an option on the group join command for the computer which sends the packet to optionally receive a copy, although in practice it seems that this particular part of the multicast standard is not the most rigorously enforced. To the uncertainties of multicast can then be added the possibilities of receiving the host's own packets, even when this has not been explicitly requested.
that unfortunate word "unreliable" in it, something that in the highly deterministic world of computing is generally considered to be a little undesirable. Before considering whether it is better to fight unreliability, or simply to accept it, it may be worth examining where the distinction between reliable and unreliable computer communications originated.

**Unreliable Communication**

Historically wide area network communication links were extremely unreliable, placing practical limits on both protocol speeds and packet sizes. In order to hide this from the host computers, communication protocols were and to a large extent still are designed to provide reliability in a noisy environment. This occurs at the expense of both bandwidth and speed, and the incurred overhead is relatively independent of the actual number of errors seen on the underlying data lines.

Because the underlying communications hardware has slowly been transformed from intrinsically noisy copper based network connections, into highly reliable fibre optic connections, data communications are now intrinsically much less unreliable. Since increasingly the causes of unreliability are not physical problems with line noise, but network load or software reliability, it can be argued that the unreliable UDP protocols are frequently not that different in reliability to the reliable TCP/IP protocols; and in the absence of such problems and relieved of the accompanying overhead they offer better performance.

In the limit, even the best communication protocol cannot be completely reliable, and this is something that no well behaved application should fail to take into account. In addition a distributed application always has to take into consideration the possibility of partial failure in its computation network, in a way that doesn’t effect single host applications. So it could be argued that using an unreliable communication protocol does not necessarily impose any additional programming burden.

There is also one distinct advantage to an unreliable protocol. Under conditions of heavy load, networks can experience non-linear performance degradations as small amounts of additional offered traffic triggers a cascade of retransmissions, as the reliable protocols retransmit timed out packets. Unreliable protocols don’t have this affect, at least they
don't automatically perform any retransmissions, and can consequently be more robust in situations of high load, provided that their applications are able to detect and resolve such situations without adding additional load to the network.

2.4.2 A multicast model for distributed computation

If the client-server model of distributed programming resulted from a star based data communications topology, what sort of model can be suggested for a multicast based system?

This thesis proposes a model where each distributed element has a peer relationship with all other elements, rather than the hierarchical relationship embodied in the client server model. Generally in distributed systems, one of the larger problems that needs to be solved is to determine which host needs to receive what data, and to see that it does so. If multicast is used as the underlying protocol, then each host in the system has to deal with the opposite case; which messages don't they want to deal with. With intelligent use of multicast groups, the scope of this problem can be restricted; forming communities of interest sorting the hosts into different multicast groups, so that they only receive data relevant to the area of the computation they are interested in.

Although each distributed element has a peer relationship with each of the other elements, it is not required that all peers be identical, either in their capabilities or their performance. Each peer is required to provide a small core set of functionality, and a mechanism for easily extending that functionality, where the local host is capable of supporting, it is also specified. This allows users to provide their own functions and programs that are run within the distributed system. Since the model is built round a message passing interface, there is also no particular programming language or operating system requirement imposed on the hosts, beyond support for multicast communications.

2.4.3 Dealing with unreliability

So far it has been argued that the unreliable protocols are now for the most part reliable enough, and that it makes sense to use them and deal with message loss as an exception, rather than incur the overhead of dealing with it as the rule. This requires that individual
peer elements be capable of detecting and dealing with this situation, but, as has been said, they should have been doing this already.

Given this basic error detection capability, unreliability can start to be treated not as a problem, but in some cases as an advantage. The application’s perspective of a distributed system, particularly one with redundant elements, is a little different from that of an application based on one computer. The distributed application has a set of requirements for each separable part of its computation, but generally as long as these are filled it doesn’t or shouldn’t care which computer actually fills these requirements. Granted it is a gross generalization to regard building reliable systems out of unreliable components as merely a problem of ensuring that there are sufficient quantities of the unreliable components, but in the case of distributed computation and components that are typically only unreliable under program error or extreme load, this is a useful description of at least part of the problem.

In the typical computer program, particularly those confined to a single machine, unreliability is normally an error condition, not design intent. Computer programs are highly deterministic creations, and although determining every single instruction set that the program could execute would typically be a very laborious process, it could in theory be done - even if it remains impossible to determine in the limit if the program would then ever halt. This is the great power of the computer program, but it is also its Achilles heel if or when the program encounters circumstances that are unexpected, at least by its designer.

2.4.4 Adaptation and distributed computing

Adaptation is generally defined as the ability to change or somehow fit actions to differing circumstances. Since the distributed networked environment that is being proposed for this thesis is inherently stochastic, adaptiveness would seem to be a useful property for any system being built to operate in such an environment.

In order to adapt to changing circumstances though, it is necessary to have some idea of what those circumstances are and have been, as well as having mechanisms that allow adaptation to take place. In a distributed message based system, adding a self-monitoring capability
to record and maintain this information is relatively straightforward, more problematic is how to react to it.

As Selfridge[19] observes, by allowing a system to make small controlled changes in a responsive environment, it is possible to improve control without having an accurate definition of the larger system. In order for this to be successful, some sense of objective or purpose is necessary to guide the changes. In evolutionary systems this objective is generally an emergent result of whatever fitness function is being imposed on the system. In a top down deterministic system, it is usually a rigidly pre-defined problem imposed goal, typically completely immutable.

A locally self-adaptive program would seem to sit somewhere between these two, while not as rigidly constrained in its activities as a non-adaptive program, there needs to be some guidelines, or rules that implicitly order its adaptation. This begs the question, what sort of rules make sense in such a system?

To begin with if a system is being allowed to make even small changes to itself, it risks making a change that will cause it to cease operating. This is highly undesirable, since it prevents any subsequent adaptation from taking place that might resolve the situation. So this suggests that the most important priority for an adaptive system should be survival, in other words it should avoid crashing, and if it does crash, it should be able to recover in such a way that it avoids repeating the actions that caused it to enter that state.

Against the negative requirement that the system will attempt to avoid crashing, there needs to be a positive requirement that the system will attempt to do something, otherwise it may simply move to a stable state where no changes are ever made - since any change would inevitably introduce the risk of a crash. This in turn raises the question of how tasks are scheduled or assigned in a distributed system in any situation where more than one computer is available, but where the computers have some choice over what they choose to do.

This thesis proposes an autonomous distributed scheduling system, rather than a centralized scheme. This is partly to avoid overhead, and also to maintain the symmetry of low overhead local decision making. In order to achieve this a mechanism is needed that allows machines
to decide whether to participate in running part of an application, based on locally obtained information only, and that also provides a reason for them to do so.

A simple market bidding mechanism is proposed, similar to that described by Malone[13]; where a machine that needs work performed by other machines broadcasts an invitation to tender, bids (so to speak), and selects from the machines that respond according to its own requirements. One of the attractions with this approach is that it allows considerable flexibility, both in the type of bidding that is used, and the values assigned to tasks throughout the larger system. This allows large scale adaptation to occur to the capabilities of the machines participating in the distributed system.

Combined with this market economy for distributed control, each local system has some limited historical information about its own performance, in terms of messages handled, which functions have been executed and their execution time, and the number of detected errors. A set of system parameters that affect behavior and performance can be dynamically adjusted in response to this information, including a minimum message receipt and sending interval, the removal of functions that are suspected of causing system instability, and dynamic recompilation and replacement of the source code for user functions.

In terms of the number of parameters that can be adjusted, and the specific areas of the system that are monitored to influence these adjustments, the adaptive capabilities of the overall system are relatively limited. It should be noted that under normal circumstances most of these capabilities are rarely used. When they are, it is in an attempt to move the system’s local and distributed state back into 'normal circumstances' - in other words, the system’s automatic adaptive behavior is confined to exceptional differing circumstances, which is, referring back to the definition of adaptation that began this discussion, probably as it should be. At a high enough level, unexceptional circumstances tend not to differ.
Chapter 3

Design Considerations

The system presented in this thesis is designed to allow a user to easily build an application spread over several computers. Kami is currently supported on Unix operating systems and requires multicast capability on the local network they are connected to. Each machine participating in the system runs a locally autonomous daemon, referred to as a kami, which communicates with its peers on other machines using the multicast protocol. Applications can use the kami infrastructure to access resources on other kami hosting machines, without needing to know specific information about resource locality or availability. Any application that can be written to use a socket or shared library interface can use the resources of a kami network.

Communication between machines is based on messaging, but all messages are mapped directly onto system or user functions. Sending a valid message to another kami has the effect of requesting that kami to execute the associated function locally, and hence is referred to as executing a function on the remote machine or machines.

Underlying this model of distribution is the assumption that each host is handling a relatively self contained part of the larger application. Kami is best suited to building coarse grained systems that consist of several applications working together on a single problem, or to sharing resources such as video feeds between several machines. Using it for fine grained applications is certainly possible, but not recommended, since the associated overhead will likely be greater than any accrued advantages.
The name kami, comes from the Japanese Shinto religion, which represents all objects as containing a kami, or spirit, responsible for that object.

### 3.1 Local Responsibility for Global Integrity

Two primary design considerations underlie how this system has been implemented. One is that no assumptions are made about inter-host communication reliability - reflecting the choice of multicast as the supporting protocol. The other is that responsibility for resolving any problems that arise from this requirement is always locally based at the most affected element of the system. For example, if a remote machine fails to respond to a request from another kami, it is always the responsibility of the machine that originated the request to resolve the situation.

Resolution is done simply by re-issuing the request, and additionally by asking for a crash check by any machines that were providing support for the machine in question. If there is no response to the re-issuance, the originating machine attempts to find another machine for the request.

### 3.2 Programming Model

The system is best suited for applications that are locally self contained. It lies in the nature of distributed computing that the smaller the granularity of the computation being distributed, the greater the consequent diseconomies of distribution that are likely to be encountered, particularly in the areas of process data sharing and synchronization. So although kami allows the user to distribute computation at the function level and above, the system is best suited for building applications where the ratio of computation performed by the local distributed elements to the amount of consequent data communication is relatively high.
3.2.1 Kami Rings

A kami ring is a specific multicast group dedicated to a task, either for a kami application or for the entire kami system. Besides the system communication ring which every kami is required to join, separate user rings can be setup which are used only by kamis working directly on a particular user application. User rings can be of two types, application command rings, which are used by the application to execute functions on other machines, and data streaming rings. Data streaming rings are treated as a special type, and have different properties from a command ring.

User rings

Each application should request and be assigned its own application command ring. Requests by the application for machines to execute its functions are then mapped by its organizing kami onto requests for machines to join this command ring, which is then used for subsequent communication solely for the application. Most of this is transparent to the application.

Data streaming rings

Data streaming rings are designed to allow large amounts of data to be sent between kamis, whilst providing the user with some protection against the idiosyncrasies of IP data communications. They are setup with a single source kami, and can be joined by any number of receiving kami. The application providing the data source should define a fixed length size for each data block that will be sent over the ring, referred to subsequently as a frame of data. For example, for a video stream, this would be the size of a single video frame from the stream, for an audio stream it would be a fixed size audio sample. The source kami must specify this fixed size for the frame size of the data it will be sending over the ring when it joins, and identify itself as the source. It can be subsequently changed if necessary.

Both receiving and sending kamis can specify a set of functions which will automatically be executed with each complete frame as a parameter. Essentially this means that the kami
system can function as a MISD\textsuperscript{1} machine under Flynn's\textsuperscript{9} taxonomy, since these functions can be different on each receiving kami.

Following execution of any application specified functions the source kami segments the data and sends it out over the ring. This can be done in one of two modes, reliable and unreliable. In reliable mode, each receiving kami passes only those packets that have been completely received to the functions that have been specified by the application. In unreliable mode, the receiving kamis maintain a frame buffer and write all received packets onto that buffer. The buffer is passed to the application functions when every frame has been received, but if packets within it have been dropped, then the frame will contain data from the current and previous frames. For certain kinds of data, raw video for example, this is a better alternative than reliable mode, which may drop whole frames of data.

3.2.2 Messaging

Kamis initially communicate by joining a predefined multicast system group, known as the system ring, and listening for messages sent on it. Within a distributed kami network messages are sent in a defined format, consisting of a function name, followed by a parameter count, and a comma separated list of parameters. Function parameters are restricted to be alphanumeric, and not include a comma, with the exception of the final parameter of any function, which is free format.

Internally separate tables of system and user functions are maintained. The system functions implement the standard set of system capabilities provided by each kami, user functions are additional functions that can be dynamically loaded and run by individual machines. Each kami maintains its own software tables, which consist of dynamically loaded shared libraries. These are initialized at run time, and can be reloaded when necessary. Application functions are provided as source code, which is then compiled by the kami. Kamis can exchange source code and recompile their libraries at any time, although in practice they will avoid doing this when an application is being run, unless severe system instability is occurring. When a message is received it is checked against the internal tables of loaded software libraries, and if a matching function is found, the function may be executed.

\textsuperscript{1}Multiple Instruction Single Data
The system currently supports libraries written in c or c++. In principle, any language can be used, provided that the language supports the standard unix dynamic library format and can handle the kami messaging interface (Appendix A, page 73).

There is never any requirement that any kami execute any function, although there is an implicit understanding that they should not refuse to do so without a reason. A number of internal mechanisms exist to provide such reasons.

### 3.2.3 Source code distribution

The simplest reason not to execute a function is that that function is not present on the local machine. Consequently the easiest way for a user to set up a kami application is to simply put the source code for the functions they will be using, only on the machines that they wish to execute the code.

Generally though, the user should as far as possible let the participating machines determine what libraries end up where, with duplication of libraries being encouraged in order to provide reliability. Each application should be organized into subsystems, with each subsystem containing at least one kami accessible function. If the subsystem includes a file `kamiprobe`, the system will use it to determine whether the underlying host can support that subsystem, before installing the function's code. Otherwise it is assumed that the function can run on any system, and if it compiles successfully it will be installed.

A kami can request the source code for functions it does not have from other kamis and can also ask other machines to check for more recent versions of the code. In order to avoid, from the user perspective at least, complete chaos; the creation of a central software repository is supported, which serves as a distribution point for all software. The distribution of code can also be restricted to be performed manually only. A graphical user interface, `qtami` shown in Figure 3-1 is provided to allow the user to examine and control source code distribution within the system.
3.3 Decision Making and Adaptiveness in a Kami Network

As has been described previously, there is no requirement for any kami to execute a function presented to it. This implies that individual kamis should have some decision making capability. Having a capability to make decisions, or choices implies that the system must have information on which to make that choice, and some set of reasons, or rules for making them. Selfridge[19] describes the importance for adaptive systems of having goals or wants, and although anarchy means an absence of government, it does not require an absence of rules.
There are as yet no good definitions of what precisely constitutes an adaptive system. Mareels\cite{15}, describes an adaptive control system as "one that is able to adapt itself to changing operations conditions", which is sufficiently vague to include most systems. In the context of this thesis, adaptiveness is used to describe the non-deterministic nature of much of kamis internal scheduling and resource allocation. Even though the mechanisms are deliberately non-deterministic, the higher level goals of a kami daemon operating within a network are clearly defined, and when choices are made by the system, it is these goals or wants that are used to govern which choice is made.

3.3.1 Self Modifying Code

Each system is built to have a self-modifying capability operating at the function level. Specifically the mapping between incoming messages and the functions that are executed as a result can be changed, allowing functions to be removed or added. Updates to the source code for these functions can be monitored, and this information is used to trigger recompilation and reloading of functions if this seems useful. This capability allows the daemons to perform a form of loose lamarckian evolution upon themselves.\footnote{Lamarck was an 18th century french scientist who proposed a scheme of evolution where organisms inherited traits which have been acquired by their ancestors during their lifetimes.}

The greatest danger for a self modifying program is to get itself into a permanently non-working state, since it can not then modify itself to attempt to solve the problem. This suggests that a good first rule for self-modifying programs is that they should under all circumstances attempt to keep themselves in a state from which they can continue to make modifications. In other words, if a kami daemon has information that suggests that a function will cause it to crash, it should avoid executing that function.

Since users are allowed to define and load functions into the system at will, this is a very necessary protection. In order to provide it, local monitoring of function performance and stability, is combined with remotely held information and restart support. Each kami will attempt to have at least one other kami agree to provide restart support in the event of a crash. The remote kami then becomes responsible for maintaining a record of crash related
information, periodically checking to see if it’s colleague has crashed, and if it has restarting the kami and supplying it with its most recent crash related information.

Individually kamis can be configured with differing ideas of their own importance, which influences the number of remote monitors they will attempt to obtain. This is done using the kami marker economy described below.

3.4 Resource Scheduling and Allocation

As a design criterion kami daemons want to execute user and system functions, since this is their entire rationale for existence. Unfortunately, the utility of executing functions is not as immediate to the system as the previous stability rule was - in fact circumstances can be envisioned when this goal would conflict directly with the first ‘survival’ goal, since new user functions could reasonably be assumed to be the most frequent source of system instability. Simply requiring the daemons to execute user functions whenever they can, limits the ability of the system to adapt to different user profiles. On the other hand, designing a system that over time will reach a collective decision not to handle any user programs because user software is ipso facto too dangerous is equally undesirable, and so a mechanism needs to be found to directly encourage the kamis to execute functions, while still giving them some autonomy when they need it.

3.4.1 The Kami Marker Economy

The internal marker economy is designed to provide a flexible market based environment for scheduling and assigning application and system tasks. Although there is no centralized list of background system tasks that need to be performed, there are system like tasks that are shared amongst the kamis, in addition to the requirements of user’s applications.

Each kami is allocated a default number of markers which it can use to pay for its system tasks. Kamis that are the source points for setting up user applications receive additional markers to fund their application requests, proportional to the number of requests they
have to make. These amounts can be adjusted to allow varying priority schemes. Typically though, system functions will be bid at a lower level than applications.

3.5 System Control and Monitoring

Allowing a system some control over its behavior implies a need for it to know what that behavior is. Within each kami there are two main sources for this knowledge. Each kami keeps performance information on functions that it has executed, which includes the time of last execution, the time taken to execute, and a frequency count. In addition there are internal monitors that periodically check for new source code files, communication issues with other machines, and that requested system actions, (such as remote monitoring), are being performed as requested.

Each monitor is paired with an associated handler, which is run to handle any conditions that are detected by the monitor. Monitors can raise alerts when their particular handler needs to be run, and the overall self monitoring function then makes decisions on when this occurs.

Both monitors and handlers can adjust the frequency that they are run, and also request changes to other handler’s frequencies.

3.5.1 Source Code Handling

Source Code Monitoring

The file monitor runs periodic checks on the local software directory for changes or additions to source code files. If these are detected, then flags are raised against the affected functions, and a flag is raised to run the file handler when appropriate. The file handler will attempt to recompile, and, if successful, the new or replacement functions will be reloaded. Information is held against each function on when the last source code update was made, and this can be used to assess code stability.
Source Code Distribution

Updates to source code can be requested by each machine, but can also be manually distributed from the central software distribution point using Qtami. If the machine has no user load, or has requested the file update, then the source code is recompiled and loaded immediately; otherwise the machine waits until these conditions are met. When a machine decides to ask for alternative function code, it does so in a two part process. First a request for source code information is made on the system ring, and is consequently seen by all machines. Requests include the name of the sub-system and optionally the file being requested, the type of operating system being used by the machine making the request, and a hash code unique to the file. Responses to the request are confined to machines of the same operating system type, unless that information was not supplied; and specify the type of system, local stability information for the function being requested, and for the kami responding to the request, and usage information for the file. If a replacement request has been raised, then the machine will attempt to get a replacement file from the most stable machine that responded. Stability is calculated as a function of the usage of the function being requested, and the responding machine's overall stability.

3.5.2 Crash Monitoring and Handling

Crash Monitoring

Crash monitoring is handled as a distributed system function. Each kami makes a system request for a least one other kami to perform crash monitoring; machines that accept this request will then periodically check that the kami process is still running on the requesting machine, and will attempt to restart it if it is not. Crash monitoring is an economic function; and so payment is included with the request, the amount of which can be modified by the requesting machine if it is not getting enough responses.

As well as requesting stability monitoring, a kami can also supply its remote monitors with system information, which it can then query on restart in order to determine the cause of the crash.
Crash Handling

The crash handler is automatically called after a remote restart occurs, and is charged with attempting to reestablish system stability, detecting the cause of the crash, and if detected, attempting to resolve the crash by finding more reliable source code. The crash handler also maintains information on local stability which is supplied to other machines on request.

The algorithm followed by the crash handler when it is triggered by a crash, is predicated on the assumption that the core kami code is not the immediate cause of the problem, although if a core function is strongly implicated in the crash it can be removed from the function tables. It is not intended that the crash handler should replace human debugging, but rather that by maintaining information on crashes their causes can more readily be identified, and that some of their more mundane causes, for example not updating all the source code files for a subsystem correctly across the distributed system, can be automatically resolved.

The model of crashing used by the crash handler divides crashes into two groups. Immediate crashes that occur every time a particular function is called, and occasional crashes, which occur semi-randomly and inconsistently, usually after the system has been running for some time, with no immediately discernable cause and often no readily discernable pattern of cause and effect. Practically speaking, immediate crashes are the only type of crash the handler is likely to detect and resolve, since diagnosing the cause of more random crashes is highly problematic.

When a crash is detected, the crash handler immediately increases the information being obtained by its crash monitors to the maximum possible, it also asks them to increases their monitoring frequency, and it may issue a request for more monitors. Any available information on the cause of the crash is then assessed for corrective measures. Available information includes any recent software changes, possibly the last function that was executed, whether the remote information monitoring has been maximized, and also stability information from other kamis.

The crash handler reacts to the local crash frequency, which is maintained as a limited length time series, and to a comparison of the crash frequency on other machines. If repeated crashes are occurring, in an identified function, then that function will be removed
from the internal tables, and marked as potentially unstable. A request will also be made
to the source code handler to find a more stable version of the software.

### 3.5.3 User Applications

**Handling of User Functions**

There are two aspects to writing a kami application. The first is breaking the application
into the parts that can be run on different machines; then these parts must adapted to be
loadable by the local daemon and accessible through that daemon to the larger system.
Generally, it is intended that each part will be defined as a kami subsystem, which is
accessed over the kami network via a set of user defined interface functions, which will
typically be replacement functions for the `main()` function of the underlying application.
These functions are required to conform to the kami messaging interface, (Appendix A),
but are otherwise unrestricted.

This raises an interesting issue. Kami is a message based system. While a user function
is being executed, no messages can be received, and so from an external perspective the
system is hung. This is a common problem in real time messaging systems, and is typically
resolved by using threads or some other mechanism for multiprocessing on the same system.³
Each kami has been deliberately kept as a single threaded program, in order to maintain
precise control over the impact of a kami daemon on the local system. There is an adaptive
element to this decision. If a kami is running a cpu intensive user function, it will be less
likely to receive incoming messages, and incur new load. In addition to this somewhat
stochastic form of load balancing, Kamis maintain internal data on average message receipt
and function execution times, and use this in conjunction with other information, when
deciding whether to volunteer for new work.

Note that the only reason this scheme works is that the underlying protocol is unreliable
- if the system fails to pick up a multicast message within the network buffer’s timeout

---

³Threads as a general term for loosely quasi parallel lightweight strands of computation, was popularized
by its use in the Mach Operating System[3]. Under the guise of various terminologies this capability has
now appeared in one form or another on most popularly available operating systems.
period, then it is simply lost. A TCP/IP connection that did this would create unfortunate repercussions at the other end of the link; for connections to a multicast group this is not a problem.

User communication facilities

Once the application has been divided into subsystems that can run on different machines, the networked application must be constructed. Applications are instantiated using system kami commands, which provide support for distributing source code within the network, requesting that user functions be run on one or more machines, explicit support for data streaming, and requests for reliability and timing considerations to the kamis handling the application.

These commands can be accessed either as ascii commands over a local socket connection to the kami, or from within a loaded user function library. Local socket access has the advantage of introducing language independence and allows applications to be constructed partially or wholly from script based languages such as perl and isis[1].

Data Streaming

Special support is provided by the system for data streaming. Only one identified kami can act as a source to a streaming ring, other kamis joining the ring will automatically receive the data sent on it. Normally kami functions are executed in response to explicit user requests, but a data streaming ring can have a number of functions associated with it, which are automatically executed every time a frame is received on the ring. This makes processing continuous streams of data, such as video content straightforward.

Support is also provided for load sharing the processing of individual frames in a data stream between multiple machines supporting the same user function.
3.6 Message Implosion

Finally, something should be said about the need to control message output rates on a distributed system, since this is a slightly larger issue than may be immediately apparent. Although the design of kami anticipates an environment where the network is faster than the computer, the capability to self-throttle message output and receipt is provided to each daemon and is actively used, primarily to ameliorate problems with host side handling of network buffers.

This acts as a real time precaution against something, that for want of a more established term, this thesis chooses to phrase message implosion. Within kami, all non-datastreaming messages are mapped directly onto functions that are executed by the local machine. These functions may also execute functions on other machines, and this creates both a computational load on the remote machines, and an associated load on the network. In general the load created by each message-function is extremely small, particularly on a gigabit network. However, in the aggregate a problem occurs if a message continually generates more traffic than it resolves, causing unlimited and potentially exponential growth in the messages being generated by the larger system.

On a single machine, a program that is continuously increasing its use of resources will eventually be manually stopped or cause a system crash. The worst that can happen is some inconvenience to other users of the machine, if there are any. In a distributed system, the same remedies apply, but the inconvenience is to all users of the network, and there may be a large number of them. Unfortunately, although it is possible for a local kami to detect that its network links are saturated or under heavy load, simply by monitoring for message loss, it is not possible for any individual kami to distinguish at any given instance between a steady state with a very high message rate, and a state where message implosion is occurring, because the larger set of machines has entered a collective state where it sources more messages than it sinks.

In fact it seems likely that message implosion is a distributed variant of the halting problem, and that it is impossible for a distributed system either locally or globally to distinguish accurately between an unbounded state of messaging growth, and one which will eventually converge.
Fortunately the unreliable nature of the underlying protocols used by kami act as some protection from the worst effects of this problem, since under continuously expanding load messages will eventually start being dropped by the network. In addition, since kami runs as a single thread, there is a tendency to natural throttling as incoming messages consume local computation time, thereby providing some limit on the number of outgoing messages.

To this can be added that from the perspective of other users of the network, a sustained heavy convergent load is no different in terms of impact from a load being caused by imploding message load; both are undesirable if they are adversely affecting the network.

So by allowing each local kami to actively monitor for message loss, and self adjust parameters that allow it to both minimize this, and maintain the rate of incoming and outgoing messages below a certain threshold, the problem can be contained, if not cured.
Chapter 4

Kami Applications.

Although kami applications are distributed over several computers, they are setup and initiated by a single kami that acts as a control point for that application, in response to user commands. Typically an application is divided into two parts; a set of subsystems loaded by the distributed kami network, and a local interface which acts as a focal point for setup and control.

Local interface commands can be sent to a kami using a direct TCP/IP socket to a designated machine, or manually using the qtami graphical user interface. Although having to connect using TCP/IP may seem burdensome, it has the advantage of being both language and machine independent, allowing considerable flexibility to the application front ends. There is also no limit on the number of local application interfaces, which can act as both control points, and as sources and destinations for data and results from the kami network.

Local interface commands sent to a kami, are mapped onto system or application kami network commands as appropriate, and routed into the network. Kami applications should be divided into subsystems, each of which runs as a relatively self-contained unit, which can be loaded by a single kami. Any functions that are called either by the local user connection, or by another subsystem, must be written to conform to the kami API, and successfully loaded by the kami running that subsystem.

There are no explicit restrictions beyond memory availability on the number of subsystems hosted by one kami, or the number of machines that are hosting the same subsystem.
However, if a local kami is unable to find the resources for a user application it will refuse to accept the job, and will close down the user socket.

Source code for user subsystems can be manually distributed to the host system using the unix filesystem, or automatically by the kami network on an on demand basis. There is also a simple graphical user interface, based on the Qt package [21], called Qtami, which allows users to specify which machines receive which source code subsystems. It is also possible to restrict subsystems to particular machines, or to machines with particular capabilities.

4.1 Considerations

4.1.1 Name Space Considerations

One of the explicit purposes of this thesis is to make it easy for users to combine several different applications into a larger distributed application. This raises the issue of what happens when there is a clash between the function names the two applications use, which for large applications is quite probable.

Since kami encapsulates applications and restricts global access to kami interface functions, applications are required only to ensure that they have unique access functions, which presents a controllable problem. There are also some advantages, especially to a system that aspires to run in a heterogeneous environment, to not imposing a completely rigid name space. Applications are free to specify subsystems with identically named interfaces that contain code customized to the underlying operating systems. This allows for example, a video feed subsystem to be written that provides a uniform interface, even though the access mechanisms the subsystem uses to obtain the video stream are operating system specific.

4.1.2 Kami file space and directory structure

As much as possible, kami attempts to use resources and systems provided by the underlying operating system. However, although there is nothing explicitly preventing the use of shared
file system facilities to store the system’s source code, this is generally not recommended. A kami should be set up with its own local file space, which is used to hold logging and long term monitoring information, and the source code of kami applications.

Each application subsystem is held in a separate directory in this file space. This allows for a great deal of flexibility on how the underlying source code for the subsystem is organized.

```
~kami
  ├── kami.log: Local Kami status and error logging
  │     ├── ltmemory/: Local kami internal long term memory information
  │     └── src/: application subsystem source code
  │         └── subsystem: video/
  │         │     ├── subsystem: cmu/
```

Figure 4-1: Kami directory organisation

Kami Interface Functions

All kami interface functions for a subsystem should be placed in the top level directory for the subsystem. The rest of the subsystem can be organized in directories below this. Each interface function must be in a separate file, which has the same name as the interface function itself. When kami loads a subsystem it scans the top level directory, and attempts to load a function for each valid source code file it finds there. The file name is used when loading the sub-system to identify and reference the interface function it contains.

When compiling a subsystem, kami looks first for a makefile, and if none is found, will compile all the files it recognizes in the top level directory of the subsystem into a shared library. If a makefile is used, kami will execute the makefile to compile the subsystem, and will then scan the top level directory for interface functions, which are then loaded from the shared library.
A kami interface function must conform to the API description in Appendix A, which essentially means that it must receive any parameters through the kami message structure shown in figure A-1, which is passed by kami to all interface functions. An API is also provided to invoke kami functions from within the application.

### 4.2 Kami Rings

Communication between different kamis takes place using kami rings, which are managed multicast groups. All kamis must join the system ring on startup, which is used for all system level communication. Only kami daemons can send messages directly over this ring.

Application functions are executed over application command rings that should be setup by the application before it requests any kami resources. Each application should have at least one command ring, this is used to communicate with machines that are running functions for that application.

Applications may also assign and use data streaming rings, which incur specialized handling for that task, and can attach functions both on the sending and receiving machines, which are automatically run on receipt of a frame from the ring.

#### 4.2.1 Ring support

Within the kami system rings are identified uniquely by their multicast address, and also by a name. The IP space for multicast ranges from 224.0.0.0 to 239.255.255.255. A kami system requires that a portion of this range be reserved for each kami’s sole use. Applications can assign addresses outside of the reserved space to their rings, in which case the onus is on the application to ensure that the address is not being used elsewhere in the system.

Normally applications should provide a name for their ring, and let the local kami system assign an address out of its own address space. This name is expected to be unique within the kami network, and can then be used as a common reference to the ring within the larger system. Most of the kami application interface functions expect a ring name, and it is also used within the larger network to map rings onto their internal data structures.
The *IP.MULTICASTLOOP* option is normally disabled on kami rings, since it is usually the case that the local kami does not want to see its own messages, and in known cases where it does it can simply execute the function directly.

For cases where the system does not want to indefinitely block on local execution, or for applications that similarly want to route messages locally, but also handle any incoming messages as they are received, it is necessary to send messages to the local machine. To resolve the problem, each kami maintains a ring with the local machine as its sole member, with the *IP.MULTICASTLOOP* option exceptionally turned on. This ring can then be used to send messages to the local machine only, using the ring name *localhost*. 
Chapter 5

Writing a Kami Application

Applications that run within a Kami network can easily do several things: real time data streaming and processing on multiple machines, combine the resources of several machines into one distinct application, and spread computational load over several computers, for example a parallel numerical calculation. It is useful to examine these categories of usage individually, in order to understand how to develop applications.

A simple example of an application from each category will be described in this chapter. Using Kami commands to setup and run an application is relatively straightforward, and the main challenge lies in designing and planning the application to run successfully in a distributed environment, and in understanding the advantages and limitations of using a Kami system to provide that environment.

5.1 Real Time Streaming: Quiz TV

Quiz TV is a simple interactive application which broadcasts questions from a quiz show over a network to up to four participants, whose pictures and answers are superimposed on the broadcast.

The resources used by the application are a database of video and audio files which hold the quiz questions, and local video resources from the machines used by the participants.
A simple set of C functions is used to tie the application together, which runs on one computer. Kami access and video processing is written in Isis[1], which uses kami user interface functions sent directly to the local kami socket interface.

5.1.1 Application sub-systems.

The Quiz TV application is an example of video streaming and resource sharing. The application has requirements for both peripherals in this case video cards, and cpu which are greater than that available on a single machine. The Compaq XP500 supports a maximum of two video cards, where this application requires between 3 and 6. At least two machines with associated video cards are required to handle the cpu requirements of providing the main data streams from disk files, and composing them together into one picture. In addition a separate video card is required to provide each user's camera feed.

The application can run on between three to seven machines, depending on the number of players. Figure 5-1 shows the distribution of program sub-systems on different machines, and their individual functions.

Figure 5-1: Quiz TV Distributed Program

QuizDB is the central repository for the quiz video files, which are accessed using the function playQuestion. Calls to this function cause the supplied question video file to be
streamed onto the question video ring. Similarly the `playerOn` initiates a video stream from the camera attached to the local workstation of the player.

Quiz Control is a set of functions that allow users to join the quiz; and to send their answers, and that sends the answers out for display on the quizviewer.

The quizviewer is an isis program which simply merges the player video and question video, and provides a merged display of their pictures and answers. Display of the answers is actually triggered by the `playQuestion` function, which automatically executes the function `answers` at a predetermined distance from the end of the question's video stream.

Once all the components of the system are in place, players can join in using a unix program which runs as a local interface program to a host kami. There is no requirement for a kami host on the machine players are participating from, since connections to the host kami are made as direct TCP/IP connections. This is another useful feature of a kami network, although all machine hosting kamis are currently required to be on the same subnet, local interface programs can run on any networked machine. This allows a core set of kami workstations to provide services well outside of their local subnet.

5.1.2 Modus Operandi

The application uses up to five data streaming rings to supply the various video components. Up to four of these are from the player workstations and are small pictures of the players, the other ring supplies the composite video feed of the background quiz program with the players video streams and answers superimposed using the isis program [1]. This is composed on a single alpha workstation and then provided to all participating workstations using a fifth kami ring. The limitation of four players is mainly cosmetic, in order not to hide too much of the original broadcast; in terms of system load, more could be supported.

Although the software parts of the system are distributed into several subsystems that can be hosted independently, there is no particular requirement to do so; kami's main contribution to this application is in handling the various video streams, and in particular allowing the video database to be housed on a separate machine to the composition machine. The video database requires considerable local storage and its own video card for format conversion.
Limitations

The main limitation on the quiz application is the performance of the composition machine. The program uses a video resolution of 320x240 pixels for its composed output, and if no other activity is present it can sustain output at near video rate of around 22 frames per second with occasional dropped packets within frames. This output rate appears to be a limitation of the workstation rather than the network. Although it is difficult to be completely sure, there are no statistics or any other indications of dropped packets or congestion indications on the network equipment being used, a Nortel Accelar gigabit router. Behavior in this and other applications suggests that packets are lost on the workstation as successive data transfers to the network card partially overwrite the communication buffers before all packets can be output.

Some amelioration of this problem is provided by the local kami pausing between successive packet writes, so that output of the packets of each frame is evenly spread within the time available.

5.2 Distributed Computation: Traffic Lights.

The traffic lights application is a simulator for exploring adaptive algorithms for traffic light timing. Each traffic light requires a single kami machine, and independently adapts the timing it uses based on purely local information, attempting to minimize the total delay it causes to traffic within the network.

The traffic light itself exists as a data structure defined and initialized within the traffic light subsystem, and traffic between lights is implemented as a single function `car`.

It can be readily noted at this point that there is no particular computational reason to write a simulation of this kind as a distributed application. Indeed one particular aspect of this simulation is particularly unsettling for a kami network, the need to use a single identified machine for each light. That said using a supporting kami network, makes it much easier and quicker to write this kind of simulation. Although this may seem wasteful of computer resources, it emulates the Boewulf system in its attitude to computers, particularly older
ones, as a cheap and plentiful resource; and uses them to considerably reduce programming effort.

This makes the simulation very flexible - extending the number of traffic lights, is simply a question of adding an extra machine.

5.2.1 Application subsystem

The application consists of two functions setLight which is used to initialize each traffic light, and car which is used to send traffic around the network. The application is initialized and run from a local kami connection made from a perl script.

Establishing the topology of the traffic light network is done using the setLight function, each light holds the next light encountered in each direction in a local array, that defines each individual light.

The car function is then used to move cars around the network. Each time a car message is received it is examined to see if it is for the local traffic light. If it is, then the function calculates whether or not it is going to be delayed, and if so by how long; and then sends itself to the appropriate traffic light by updating its parameters and sending itself.
So in effect, each car in the simulation is represented by a message in the kami network that is continuously updated and passed between traffic lights as it moves through green and red lights. Whilst it is doing this, the car function also executes an adaptive algorithm that changes the red-green intervals on the local traffic light, allowing the simulation to explore different ways of adapting to traffic load.
Source code for kami user function car.c

```c
#include <stdio.h>
#include <time.h>
#include "traffic.h"

int initialised = -1; /* flag on this machine's init status */
struct light trafficLight;
int north = 0;
int east = 1;
int south = 2;
int west = 3;

/* Function : car(struct msgStruct *msg) */
char *car(struct msgStruct *msg)
{
  int id;
  int total_delay;
  int routeIndex;
  int numberOfRoutes;
  int *route;
  int queued;
  int direction;
  int finished;
  int t;
  int lightId;
  int i;
  int delay;
  int red;
  int sequence;

  /* Get parameters for command */
  id = atoi(getParam(msg, 1));
  lightId = atoi(getParam(msg, 2));
  sequence = atoi(getParam(msg, 3));
  total_delay = atoi(getParam(msg, 4));
  queued = atoi(getParam(msg, 5));
  routeIndex = atoi(getParam(msg, 6));
  numberOfRoutes = atoi(getParam(msg, 7));

  /* Work out the state of the lights for this car */
  while(!finished)
  {
    if(currTime < t + trafficLight.interval_north)
    {
      finished = TRUE;
      red = north;
      delay = t + trafficLight.interval_north - currTime;
      continue;
    }
    t += trafficLight.interval_north;

    if(currTime < t + trafficLight.interval_east)
    {
      finished = TRUE;
      red = east;
      delay = t + trafficLight.interval_east - currTime;
      continue;
    }
    t += trafficLight.interval_east;

    /* The first time the car arrives at this light, the light computes */
    currTime = trafficLight.baseTime;
    /* See if this car is at this traffic light */
    if(trafficLight.id != lightId)
      return (char *)NULL;
    
    /* Read routes for command */
    route = (int *)(malloc((sizeof(int *) * numberOfRoutes)));
    for(i = 0; i < numberOfRoutes; i++)
    {
      route[i] = atoi(getParam(msg, 8 + i));
    }
    direction = route[routeIndex];
    finished = FALSE;
    t = trafficLight.lastMsgTime;
    if(trafficLight.interval_north == 0 ||
      trafficLight.interval_east == 0)
    {
      printf("Urgh, stuck on red - resetting!");
      trafficLight.interval_north = DEFAULT_INTERVAL;
      trafficLight.interval_east = DEFAULT_INTERVAL;
    }
    delay = 0;
    
    /* Work out the state of the lights for this car */
    while(!finished)
    {
      if(currTime < t + trafficLight.interval_north)
      {
        finished = TRUE;
        red = north;
        delay = t + trafficLight.interval_north - currTime;
        continue;
      }
      t += trafficLight.interval_north;

      if(currTime < t + trafficLight.interval_east)
      {
        finished = TRUE;
        red = east;
        delay = t + trafficLight.interval_east - currTime;
        continue;
      }
      t += trafficLight.interval_east;
    }
  }
  
  return (char *)NULL;
}
```

Source code for kami user function car.c
if(queued != TRUE)
{
    if( ((direction == north) || (direction == south)) && (red == north))
    {
        total_delay += delay;  // delay seen by this car
        trafficLight.total_delay += delay;  // delay caused by this light
        /* For a very simple algorithm - try reducing red time for this */
        /* direction. */
        if(red == north)
        {
            trafficLight.interval_north /= 0.8;
        }
        else if(red == east)
        {
            trafficLight.interval_east /= 0.8;
        }
        else
        {
            printf("Bad direction in car.c = %d", direction);
        }
        time(&trafficLight.lastMsgTime);
        trafficLight.lastMsgTime -= trafficLight.baseTime;
        setParamInt(msg, 6, total_delay);
        queued = TRUE;
    }
    else
    {
        /* So if we're going through on green - increase the time for this */
        /* direction. After all, there should always be symmetry */
        if((direction == north) || (direction == south))
        {
            trafficLight.interval_north += 1.2;
        }
        else if((direction == east) || (direction == west))
        {
            trafficLight.interval_east += 1.2;
        }
        else
        {
            printf("Bad direction in car.c = %d", direction);
        }
        if(routeIndex >= noRoutes) routeIndex = 0;
        else routeIndex++;
        /* set new traffic light id. */
        setParamInt(msg, 2, trafficLight.map[direction]);
        setParamInt(msg, 6, FALSE);
        setParamInt(msg, 6, routeIndex);
    }
    if(atoi(getParam(msg, 2)) == trafficLight.id)
    {
        /* moving to self/this light - until queued == true */
        sendKamiMsgToSelf(msg);
    }
    else
    {
        setParamInt(msg, 3, ++sequence);
        sendKamiMsgToRing(msg, "traffic");
        printf("Moving to Light Xs seq Xs Total delay = Xd local delay = Xd Time = Xd/\m", \
               getParam(msg, 2), getParam(msg, 3), total_delay, delay, currTime);
    }
    free(route);
    return (char*)NULL;
}
Source code for setLight.c and traffic.pl

```c
#include "traffic.h"

extern struct light trafficLight;
extern int initialised;

char *setLight(struct msgstruct *msg)
{
    trafficLight.id = getAppIdO;
    if(atoi(getParam(msg,1)) == trafficLight.id)
    {
        trafficLight.map[0] = atoi(getParam(msg,2));
        trafficLight.map[1] = atoi(getParam(msg,3));
        trafficLight.map[2] = atoi(getParam(msg,4));
        trafficLight .map[3] = atoi(getParm(msg,5));
        trafficLight.interval.north = DEFAULT_INTERVAL;
        trafficLight.interval.east = DEFAULT_INTERVAL;
        time((trafficLight.baseTime);
        printf("Setting routes for trafficLight ", trafficLight.id);
    }
    return (char *)&NULL;
}
```

```perl
use IO::Socket;
my $sock = new IO::Socket::INET ( PeerAddr => 'localhost',
    PeerPort => '6014',
    Proto => 'tcp',
) ;

die "Could not create socket: $!" unless $sock;
print $sock "local:<assignApplicationRing,traffic>"
print $sock "local:<assignFunction, setup, 4, traffic, traffic>"
sleep 1;
print $sock "traffic:<setLight, 15, 12, 11, 12, 11>"
print $sock "traffic:<setLight, 11, 14, 15, 14, 16>"
print $sock "traffic:<setLight, 14, 11, 12, 11, 12>"
print $sock "traffic:<setLight, 12, 16, 14, 15, 14>"
print $sock "traffic:<car,1,15,1,0,0,0,4,1,2,3,0>"

close($sock);
```
5.2.2 Observations

This application uses a kami network to minimize programmer work, rather than for any intrinsic reasons of computational necessity. However it does demonstrate how easy it is to setup a realtime simulation with a kami network, which exploits a distributed network, and there are many areas, in particular those involving genetic algorithms, where such simulations do require considerable local computation but could readily take advantage of a distributed environment.
Figure 5-3: Output from the Traffic Light Simulation as it converges to gridlock.
Chapter 6

Future Directions

Kami has been deliberately designed to have a small, easily ported core of functionality which can be readily extended, by adding additional user or system sub-systems. In principle the only requirement for a computational device to participate in a kami network is support for some kind of TCP/IP communication.

This suggests several areas for expansion in the event that some value is found in writing distributed applications using a kami network for support.

6.1 Support

Kami is currently supported under the Digital Unix and Linux operating systems. Providing kami daemons that run on small computational devices such as PDA’s, would present interesting opportunities for interaction with applications running in a larger kami environment, and should be relatively straightforward.

6.2 Security

Making a secure distributed system was not tackled as part of this thesis, although the requirements for security were considered, in order to ensure that it would be possible to add this feature if it became necessary.
Kami relies on the security of the operating system it is running on. Normally it is expected that a kami daemon would be run with its own directory space, and it is suggested that a dedicated user id also be used. Giving a kami daemon local root privilege would probably not be a good idea. Taken together, this implies that at present a kami network can be considered to have the same amount of security as that of the least secure machine running a participating daemon. In passing, it can be noted that it is unlikely that running a kami daemon increases the risk to other machines in the network, since once a packet sniffer is being run on a network, there are easier ways to penetrate the other machines than spoofing internal kami commands.

However, if kami were ever to be run in a non-research environment, security would have to be a major concern. The architecture deliberately makes adding security a relatively easy task. The two key areas to augment would be encryption, and associated identification signing, of all kami system and application commands sent over the multicast rings.

In addition the method used to make source code changes should be extended to include checking, verification and logging of all updates. It should be noted that dynamic code changes are not an absolute requirement of a kami network, and the functionality is provided as a dynamic sub-system, making it easy to exclude if required.

\section{Wide Area Operations}

In most networking environments communication using multicast groups is confined to a single subnet. How well a kami network scales to a large number of systems is partly a function of the application being run, and its demands on the network, and partly determined by the behavior of the throttling behavior that kami uses to restrict network load. At present this is done very simplistically, and would need to be revisited in a wide area network, where much more attention would need to be given to relative speeds and latencies between machines.

This however assumes continuous use of multicast within the system. Another area worth exploring, particularly suited for linking up local clusters of systems over a wide area network would be to develop a \textit{bridge} subsystem, which would setup dedicated TCP/IP based
communication links between two separated kami networks. This would allow switch imposed limitations on the scope of local multicast packets to be circumvented, and would also provide well defined control links between such systems. It is of course, an interesting question what kind of application could be developed to exploit such an environment; although some obvious issues with redundancy and traffic load would need to be carefully evaluated.

6.4 Scaling

Scaling is an unexplored area of this thesis; the largest number of machines that have participated in a kami network is currently five kami hosts and four local machines. The behavior of a large kami network running several user applications concurrently would necessarily be dependent on the type of applications, since the load placed on the network by kami system traffic is relatively small. Most interesting perhaps, would be to see how a large kami network would be successful in managing itself under conditions of heavy load to maintain reliability throughout the system. This would also allow greater exploration of the message implosion problem, and a determination of how critical a problem this is in practice for generalized distributed applications.
Appendix A

Kami Application Interface

Kami applications are typically written as a mixture of local kami interfaces which communicate directly with the kami daemon on the local host, and subsystems which are loaded into kamis running on one or more hosts, and are accessible only through the kami network. Local interfaces use a TCP/IP socket connection to the local kami, and can be written in any language or application that supports this form of communication.

Both local interfaces and subsystems communicate with the kami network by making requests for functions to be executed, either on the local host, or on the larger kami network. Requests issued by local interfaces are mapped onto the appropriate set of system commands, and initially sent by the local kami over the system control ring to other machines. Usually the first thing an application will do is request an application ring and an associated number of machines, which is then used to issue requests specifically for the application’s sub-systems.

There is no restriction on the number of local interfaces involved in a single application. Local interfaces can act as data sources and sinks, as well as making requests to the kami network. For example the video subsystem for a Compaq Alpha which sets up a video stream on a kami ring, uses a local interface to send the video data to the local kami which then multicasts to all machines on the requested kami ring. However, a kami subsystem function is provided to control access over the kami network to the video resource, and it is this that runs the local interface.
All kami interface functions, that is functions which will be directly loaded and accessible to other kami systems, must conform to the kami messaging interface. The standard template for these functions is:

\[
\text{char *interfaceFunction(struct msgStruct *msg)}
\]

Interface functions are declared with one parameter, a pointer to a message structure, which is used to pass function parameters. The kami messaging interface consists of this structure, parameter access functions, and functions that allow kami messages to be sent to other machines. The return parameter is a free format string, which is not used by kami, but can be accessed by other local user functions.

### A.1 Interface Functions

#### A.1.1 Parameter Passing

Each interface function is passed a message structure, which contains the function’s parameters. All members of the structure can be accessed directly, but utility functions to do so are also provided. Function `getParam()` is provided to directly return the parameters to an interface function. All parameters are passed as ascii strings, and it is up to the interface function receiving the message to correctly decode them.

```
struct msgStruct
{
    char fromIP[IP_ADDR_LENGTH];    /* ip address of sending host */
    int fromSocket;                 /* socket no. of sending socket. */
    char function[MSG_FUNCTIONL];   /* name of function being sent */
    char toRing[MAX_NAME];          /* ring function is to be exec'd on */
    int noparams;                   /* no. of parameters in list */
    struct parameter *param;       /* list of parameters on this msg */
    time_t receivedTS;             /* time message received */
    struct socket_def *sockp;      /* pointer to socket structure msg */
    char ackFlag;                   /* flag on msg being received and ackd*/
    int msgNo;                      /* sequence no. of this message */
};
```

Figure A-1: Kami Parameter Structure
The message structure includes information identifying the host that originated the message, and the name of the kami ring it was received on, as well as a list of the parameters sent to the function. These will be automatically set by the underlying kami structure. The message structure can be updated and re-used by the application if it needs to.

A.1.2 Example of Interface Function

Figure A-2 shows a sample interface function which, if it were part of a kami subsystem, would be copied to the top level of the subsystem directory structure, in a file called interfaceFunction.c.

```c
#include "kami.h"

char *interfaceFunction(struct msgStruct *msg)
{
    char *parameter_1;
    char *parameter_2;

    int x;
    int name;

    parameter_1 = getParam(msg, 1);
    parameter_2 = getParam(msg, 2);

    x = atoi(parameter_1);
    strcpy(name, parameter_2);

    /* And do something with the data */
}
```

Figure A-2: Kami Interface Function Source Code

A.2 Local Socket Interface

In order to setup and control an application running within a kami network, a local socket interface is used. The commands sent over this interface are the same as those available to
the application as functions. The format for these commands is in ascii, and is specified as follows:

```
command scope:<function name, (comma separated parameter list)>
```

Command scope identifies how the supplied function is to be handled: `local` indicates a command for the local kami, typically a setup, initialization or cleanup function; `system` is a command for the kami system ring, and otherwise the name of the application ring that the command is to be sent on should be used.

The function name should be exactly as specified in the source code for the function, and is case sensitive.

### A.2.1 Example

```perl
use IO::Socket;
my $sock = new IO::Socket::INET ( PeerAddr => 'localhost', PeerPort => '6014', Proto => 'tcp', );
```

die "Could not create socket: $!
" unless $sock;

```perl
# Assign the application command ring called traffic.
print $sock "local:<assignApplicationRing,traffic>";
```

```perl
# Ask for four machines to join the ring with the application # functions, setup, car and setLight.
print $sock "local:<assignFunction, setup, 4, traffic>";
print $sock "local:<assignFunction, car, 4, traffic>";
print $sock "local:<assignFunction, setLight, 4, traffic>";
```

```perl
# Run the application.
print $sock "traffic:<setLight, 1, 3, 2, 3, 2>";
print $sock "traffic:<setLight, 2, 4, 1, 4, 1>";
print $sock "traffic:<setLight, 3, 1, 4, 1, 4>";
print $sock "traffic:<setLight, 4, 2, 3, 2, 3>";
print $sock "traffic:<car,10,1,0,0,0,4,1,2,3,4>";
print $sock "traffic:<car,11,2,0,0,0,4,1,2,3,4>";
print $sock "traffic:<car,12,3,0,0,0,4,1,2,3,4>";
```

```perl
close($sock);
```

Figure A-3: Setup and Control Application perl script

Figure A-3 shows a perl script which sets up a kami application, using a socket connection to a local kami. A single application called traffic is requested, and a request is made for four machines to run the application. Appendix B lists all the available interface functions.
Appendix B

Kami Interface Functions API

The application programming interface can be broadly divided into three parts: utility functions which provide support for creating and handling kami functions and rings, commands for sending kami functions or commands to other kami daemons to be executed; and a set of system provided kami functions.

For application purposes there is no difference between the way the system provided kami functions are handled and those installed by the application, but their source code is not generally accessible, with the exception of some system packages, such as the software central feature which are provided as optional subsystems.

B.1 Sending Kami Commands and Messages

In terms of their affect a kami command and a kami message are identical. For the local interface, a distinction is made between commands that send a prepared kami message (deprecated, unless re-using an existing message), and those that take the command parameters and format and send a kami message.
Table B.1: C and C++ support for sending Kami Functions and Messages

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Parameters</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Creating and sending kami commands</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sendKamiCmd</td>
<td>1 Socket</td>
<td>int</td>
<td>socket number to send command to.</td>
</tr>
<tr>
<td></td>
<td>2 Ensure Delivery</td>
<td>char</td>
<td>Indicates whether message delivery should be confirmed: true = 1 false = 0</td>
</tr>
<tr>
<td></td>
<td>3 Function name</td>
<td>char *</td>
<td>Name of function to send/execute</td>
</tr>
<tr>
<td></td>
<td>4 No. of parameters</td>
<td>int</td>
<td>No. of parameters being supplied to function</td>
</tr>
<tr>
<td></td>
<td>5 Function parameter</td>
<td>char *</td>
<td>Function parameters in char * format</td>
</tr>
<tr>
<td>sendSingleKamiCmd</td>
<td>1 Socket</td>
<td>int</td>
<td>Socket number to send command to.</td>
</tr>
<tr>
<td></td>
<td>2 Ensure Delivery</td>
<td>char</td>
<td>Indicates whether message delivery should be confirmed: true = 1 false = 0</td>
</tr>
<tr>
<td></td>
<td>4 Function name</td>
<td>char *</td>
<td>Name of function to send/execute</td>
</tr>
<tr>
<td></td>
<td>5 No. of parameters</td>
<td>int</td>
<td>No. of parameters being supplied to function</td>
</tr>
<tr>
<td></td>
<td>5 Function parameter</td>
<td>char *</td>
<td>Function parameters in char * format</td>
</tr>
<tr>
<td><strong>Sending kami messages</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>createKamiMsg</td>
<td>1 Function Name</td>
<td>char *</td>
<td>Name of function</td>
</tr>
<tr>
<td></td>
<td>2 No. of parameters</td>
<td>char *</td>
<td>No. of parameters being supplied for function.</td>
</tr>
<tr>
<td></td>
<td>5 Function parameter</td>
<td>char *</td>
<td>Function parameters in char * format</td>
</tr>
<tr>
<td>sendKamiMsg</td>
<td>1 Message to send</td>
<td>struct msgStruct *</td>
<td>Pointer to message structure</td>
</tr>
<tr>
<td>sendKamiMsgToRing</td>
<td>1 Message to send</td>
<td>struct msgStruct *</td>
<td>Pointer to message structure</td>
</tr>
<tr>
<td>Function Name</td>
<td>Parameters</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------------</td>
<td>------------</td>
<td>--------------</td>
<td>------------------------------------</td>
</tr>
<tr>
<td>Send command to specified ring</td>
<td>2</td>
<td>Ring name</td>
<td>Name of ring to send message to</td>
</tr>
<tr>
<td>sendKamiMsgToSelf</td>
<td>1</td>
<td>Message to</td>
<td>Pointer to message structure</td>
</tr>
<tr>
<td>Send a message to the local machine</td>
<td></td>
<td>struct msgStruct</td>
<td>Pointer to message structure</td>
</tr>
<tr>
<td>sendKamiMsgToAll</td>
<td>1</td>
<td>Message to</td>
<td>Pointer to message structure</td>
</tr>
<tr>
<td>Send a message to the local machine</td>
<td></td>
<td>struct msgStruct</td>
<td>Pointer to message structure</td>
</tr>
<tr>
<td>and the named ring</td>
<td>2</td>
<td>Ring Name</td>
<td>Ring to send to</td>
</tr>
</tbody>
</table>
## B.2 Utility Functions

Table B.2: C and C++ utility functions

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Parameters</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getParam</td>
<td>kami message</td>
<td>struct msgStruct *</td>
<td>Pointer to kami message</td>
</tr>
<tr>
<td>Get a parameter from a message</td>
<td>1 Number</td>
<td>int</td>
<td>Number of parameter to fetch, first = 1</td>
</tr>
<tr>
<td>setParam</td>
<td>Kami message</td>
<td>struct msgStruct *</td>
<td>Pointer to kami message</td>
</tr>
<tr>
<td>Set an existing parameter to a new value.</td>
<td>1 Parameter No.</td>
<td>int</td>
<td>Number of parameter to set</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3 New value</td>
<td>char *</td>
<td>New value for parameter</td>
</tr>
<tr>
<td>setParamInt</td>
<td>Kami message</td>
<td>struct msgStruct *</td>
<td>Pointer to kami message</td>
</tr>
<tr>
<td>Set an existing parameter to a new integer value.</td>
<td>1 Parameter No.</td>
<td>int</td>
<td>Number of parameter to set</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3 New value</td>
<td>int</td>
<td>New value for parameter</td>
</tr>
<tr>
<td>getRingSktNo</td>
<td>Ring Name</td>
<td>char *</td>
<td>Name of Ring</td>
</tr>
</tbody>
</table>
B.3 Kami System Commands

Kami system commands act as requests to the larger system, i.e. one or more of the kami hosts, to setup and provide the application's distributed environment. They provide support for assigning multicast rings, asking machines to join multicast rings if they are prepared to host a requested subsystem for the application, and closing down the application.

Note that system commands are not c or c++ functions per se, but have to be sent to the kami system using one of the sendKamiCmd functions. Also all parameters to system commands must be supplied as strings.
Table B.3: Kami System Commands

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Parameters</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>assignApplicationRing</td>
<td>1</td>
<td>ring</td>
<td>Name of ring to assign</td>
</tr>
<tr>
<td>Assigns a ring for this application</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>assignFunction</td>
<td>1</td>
<td>fn</td>
<td>Name of function to assign</td>
</tr>
<tr>
<td>Request for N machines to run the specified function. These machines will then join the supplied ring.</td>
<td>2</td>
<td>num</td>
<td>No. of machines being requested for function</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>sys</td>
<td>Name of function’s sub-system</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>ring</td>
<td>Name of ring to assign function too</td>
</tr>
<tr>
<td>assignSingleMachine</td>
<td>1</td>
<td>fn</td>
<td>Name of function to assign</td>
</tr>
<tr>
<td>Request for a specific machine to join a ring. Should only be used for resources, eg. video feeds that are tied to a specific machine.</td>
<td>2</td>
<td>sys</td>
<td>Name of function’s sub-system</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>ip</td>
<td>IP address of required machine</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>ring</td>
<td>Name of ring to assign function too</td>
</tr>
<tr>
<td>joinMultiCast</td>
<td>1</td>
<td>ip</td>
<td>IP address of ring to join</td>
</tr>
<tr>
<td>Request to receiving machine to join a ring. Normally sent on an application ring, in order to request attached machines to join a data streaming ring. Use sendSingleKamiCmd to send to source machine</td>
<td>2</td>
<td>port</td>
<td>Port number for ring</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>dir</td>
<td>Either send or receive.</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>type</td>
<td>stream or command</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>name</td>
<td>Name for ring</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>reliable</td>
<td>no = streaming rings overwrite frames</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>local</td>
<td>local socket no. to receive data</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>frame</td>
<td>Required if direction = send</td>
</tr>
</tbody>
</table>
Bibliography


