A Peer-to-peer Message Passing System for Parallel Computing

A thesis presented
in partial fulfilment of the requirements
for the degree of

Master of Science
in
Computer Science

at Massey University, Albany,
New Zealand.

Enrico de Klerk

2004

Abstract

This thesis presents an implementation of a computational grid system that utilises network-enabled computers to execute parallel applications. The system is fully decentralised and self-configuring and handles the joining and departure of nodes transparently to the user. The system allows users to specify the resources such as operating system, network connection speed and system memory their applications require. It then searches for these resources and executes the applications on appropriate peers. It uses checkpointing of application processes to allow a running process to vacate a host and migrate to another peer when the host leaves the network. Process migration is transparent to the user and the processes automatically find the new address of the migrated process when migration is completed.

Acknowledgements

I would like to thank my supervisor, Dr. Chris Messom, for his guidance, patience and numerous good suggestions during the past year. I am also grateful to Andre Barczak for introducing me to parallel computing and his enthusiasm for the subject.

Finally, I would like to thank my parents, Colin and Elsa de Klerk, for their endless support and encouragement throughout my life.

Contents

List of Figures

List of Tables

Introduction

Parallel and distributed computing systems are continually becoming more popular for implementing and sharing resources. Grid computing is a method of exploiting the power of many computers on a wide area network to produce a distributed parallel system that dynamically shares resources to increase performance and availability.

Many of the current grid computing implementations use the client/server model where a single or a few centralised servers control many clients; this has some disadvantages. The server is a single point of failure, if the server crashes the whole system is unavailable. The server needs to communicate with all clients, this creates a large amount of traffic that is sent to and from server, which could overwhelm even a fast network connection. Other parallel computing implementations require an administrator to configure the system every time the system resources change.

A peer-to-peer approach overcomes many of the disadvantages of these approaches. It is more robust than the traditional client/server model. If a peer is lost, the rest of the system can continue working. All the peers act in exactly the same way and can communicate among themselves. This allows communication to be distributed throughout the network instead of concentrating it at a few servers. However, the wide area distribution and potentially slow communication of the system makes it best suited to programs that need little or no communication during execution.

Chapter 1 covers many of the components that are required to build a networked parallel computing system. Topics covered include resource discovery, resource allocation and process migration. The implementation of a peer-to-peer message passing system is discussed in chapter 2 and the performance of the system is compared with that of MPICH in Chapter 3. Possible further improvements to the system are discussed in Chapter 4 that are necessary for a complete parallel computing infrastructure.

1 Literature Review

1.1 Resource discovery

A system that is constantly changing due to resource failures and temporary availability of other resources requires an infrastructure that allows the user and user programs to locate the resources they require. Such an infrastructure needs a way to describe the resource, and a service that maps the resource description to its location in the system.

1.1.1 Resource Specification

Globus [8] uses the Resource Specification Language to express resources requirements by defining sets of parameter names and values. Resource brokers translate general high-lever RSL specifications to specialised RSL specifications where the locations of resources that meet the requirements are specified.

Condor uses the ClassAd Framework [1] , which defines a description language for specifying host resources such as the operating system, amount of memory, load average etc. as well as the resource requests of the application. The framework also supports attributes such as groups to allow users to specify preferred clusters.

1.1.2 Resource lookup

Metacomputing Directory Service

The Metacomputing Directory Service (MDS) [7] provides a standardised framework for managing information about the grid, its resources and the state of these resources. This allows applications to automatically configure themselves where

manual configuration is not feasible and carefully select resources, algorithms, and networks to use the grid as effectively as possible.

MDS uses an object-oriented approach to information representation, where information is an instantiation of an object class such as an organisation, person, network or computer. Each instantiation contains the unique name (called the distinguished name) of the object as well as one or more attributes and their corresponding values. MDS represents objects within the hierarchical Directory Information Tree where the path from the root to the object forms its distinguished name. Objects at the same level of the Directory Information Tree must have at least one attribute that distinguishes it from other objects at the same level.

A Class in MDS is created by specifying attributes and the type of values they may contain. Attributes are defined as either optional or mandatory. Existing classes can be extended by using inheritance to add attributes to the classes.

Grid Resource Information Service

GRIS [8] collects information about a specific resource and provides a uniform interface for querying information about their current configuration, capabilities and status. A GRIS usually reports its information to a Grid Index Information Service using the Grid Resource Registration Protocol (GRRP). The Grid Index Information Service (GIIS) [8] combines information from multiple GRIS services to produce a unified system view.

Peer-to-peer resource discovery

This section presents three different algorithms: Content Addressable Network, Chord and Pastry. The peer-to-peer networks allow nodes in the network to advertise their resources and search for other nodes' resources. The Content Addressable Network (CAN) and Chord algorithm implementations are discussed in section 2.2 and compared in section 3.1.

Content Addressable Network

CAN [10] is a peer-to-peer object location algorithm that represents a network as a Cartesian coordinate space that wraps around and has at least two dimensions. Each node controls a section of coordinate space called a zone. A hash function is used to generate the appropriate coordinate for an object; the coordinate is then stored at the node that handles the zone together with the IP address of the node where the object resides. The owner of an object periodically updates its key value pairs with the node that controls the key's zone to ensure that the network is up to date. When a node searches for a key it sends a request in the direction of the appropriate zone, each node forwards the request to its neighbour until it reaches the requested zone node that does a lookup and returns the address of the object owner.

A node joins the network by first contacting a bootstrap node, which is a node associated with a known DNS name and has a list of some of the active nodes in the network. The joining node then randomly selects a point P in the address space and sends a join message to that point. The node N that controls the zone containing P splits its zone into two equal pieces and offloads one half onto the new node together with the key value pairs contained by the zone after which N sends its neighbour table to the joining node. The joining node uses the N neighbour table to construct its own table and replaces the entry of N's neighbour on the opposite side of N with N's address; N also updates its table by replacing the node that is no longer its neighbour with the joining node. Next N and the joining node notify their neighbours of the changes to allow them to update their neighbour tables.

A node departs from the network by notifying its neighbours of its departure and handing its zone as well as its key values to one of the neighbours. The neighbour node will then either merge the departing node's zone with its own or find another node to merge with the zone. If a node stops responding or leaves the network without notifying the network each node starts a timer as soon as it detects the failure. When the node's timer expires, it sends a TAKEOVER message containing its zone size to the lost node's neighbours, when a node receives a TAKEOVER message it resets its own timer if the message contains a smaller zone size than the local size, otherwise it replies with its own TAKEOVER message. This mechanism ensures that the node with the smallest zone claims the zone of the failed node. The lost key value

pairs of the failed node are slowly repaired as the object owners periodically send updates to the zone controlling node.

Figure 1-1 a) shows an example of a network where node 2 wants to leave the network. Node 2's zone was formed when it was divided between it and node 4. Node 4's zone was subsequently divided between nodes 4 and 5. After which node 5 gave half of its zone to node 6 when it joined the network. In the figure, the grey node indicates the node that is looking for a node with which to merge and the group of black nodes indicate the zone needed to merge the zones. Node 2 first tries to merge its zone with node 4's zone, which took half of node two's zone when it joined, however this is not possible as the nodes have different zone sizes. To find a pair of nodes that can merge it sends a message to node 4.

In Figure 1-2 b) Node 4 tries to merge with zone five but cannot because node 5 shared its zone with node 6 so it forwards the message to node 5. In Figure 1-2 a) Node 5 has the same zone size as node 6, which allows them to merge. Node 6 incorporates node 5's zone with its own and node 5 takes over the departing node's zone. Figure 1-2 d) shows the network after node 6 has incorporated node 5's zone into its zone, node 2 has left and node 5 has taken over its zone

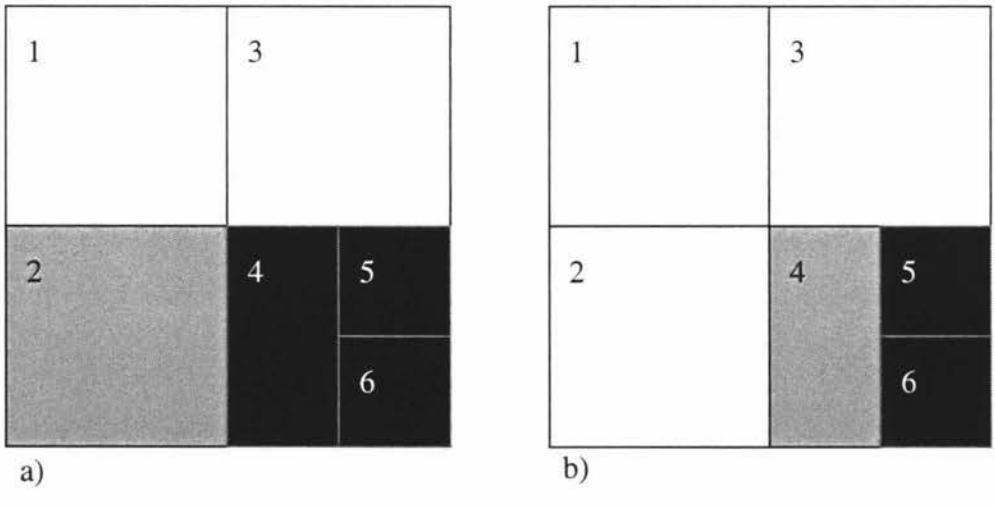

Figure 1-1 Merging of CAN zones during departure

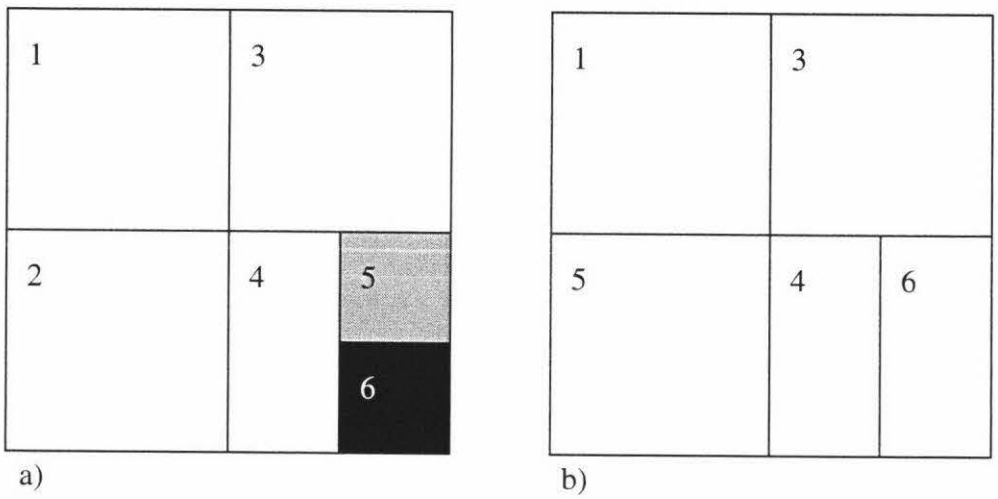

Figure 1-2 Merging of CAN zones during departure

The CAN network allows the use of multiple realities where each reality is a copy of the network coordinate space but each node is responsible for controlling a different zone for every reality. This increases fault tolerance as each zone is replicated on multiple nodes. It also improves routing performance because a node can select the reality to use where the destination node is closest. Multiple realities increase the storage necessary to keep track of all the neighbours in all the realities by O (r) where r is the number or realities.

The number of dimensions and realities used in the network determines the number of neighbours a node needs to keep track of, but stays constant as the network grows and shrinks but the average number of routing hops is affected by the network size. More dimensions in a CAN improve routing efficiency while more realities improve routing performance as well as fault tolerance but increase the overhead of keeping the network alive. The average path length of a CAN is O ($n^{1/d}$) where d is the number of dimensions and n is the number of nodes

Pastry

Pastry [11] is a peer-to-peer routing and object location algorithm that uses a locality heuristic to improve routing performance. The maximum number of routing hops is $\log_2 bN$ when there are no node failures along the routed path. Each node is associated

with a unique 128-bit node ID that represents the node's location in a circular network. The IDs are evenly dispersed among the nodes by using methods such as generating a random ID for each node or by calculating a hash value from a unique identifier such as the node's IP address.

Every node keeps track of a leaf set that contains the L nodes with the numerically closest IDs of which L/2 have smaller IDs and L/2 have larger IDs where L is typically 2^b or 2^{b+1}. Each node builds a routing table with $\log_2 bN$ rows and $2^b - 1$ columns of which row n of the table contains node IDs that have the a shared prefix which means that the first n digits as the local node ID as well as a neighbourhood set that contains IDs of nodes that are geographically close to the node. The value of b is a compromise between the size of the neighbourhood, leaf and routing tables and the average routing distance.

In Figure 1-3 the flowchart shows the steps in Pastry routing. During routing, a Pastry node first checks if the message key is within the range of the node IDs in the Leaf set in which case it forwards the message to the destination node or the node with the ID closest to that of the message key. If it is not in that range, the node forwards it to a node in the routing table that shares a prefix that is at least one bit longer than the shared prefix of the current node. If the routing table does not contain an appropriate entry or the destination node is unavailable the message is forwarded to a node whose ID shares a prefix with the message key that is at least as long as that of the current node but is numerically closer to the destination node.

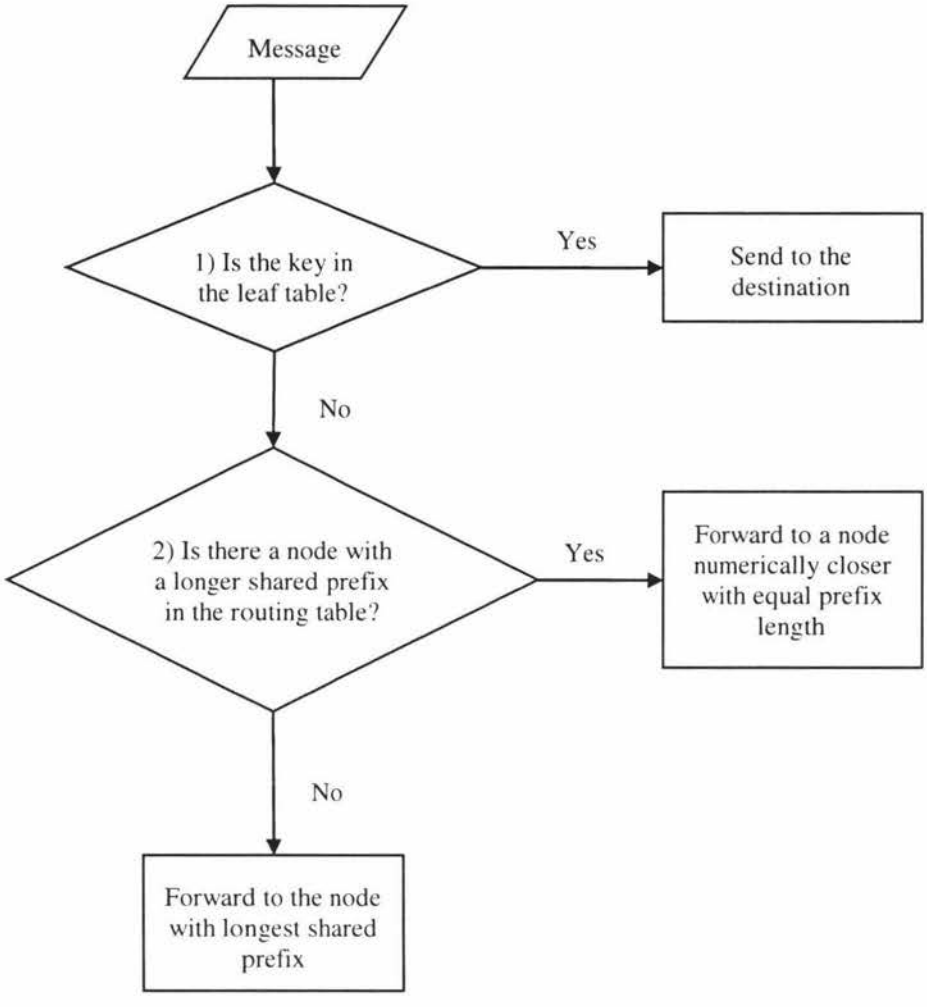

Figure 1-3 Pastry routing flowchart

A new node connects to an existing network by connecting to a nearby node by doing an expanding ring IP multicast or selecting an address from a preconfigured list and sends a join request message containing its own ID as the key. The request is then routed to the node with the ID closest to the message key. As each node along the path to the destination receives the join request, it sends its state tables to the joining node. The joining node uses this information to create its own neighbourhood set, routing table, and leaf set. It can also request additional state tables from other nodes to fill its own tables. After the joining node has completed its state tables it notifies all the affected nodes of its presence and finally sends its state tables to all the nodes contained in the tables to allow them to update their tables. If a node detects a failed

node in the leaf set it requests the leaf table of the node with the highest node ID in the local leaf set on the side of the failed node. It then selects a replacement node from the received leaf table.

Chord

Stoica *et al.* [17] describe a simple peer-to-peer network that maps a key to a node on the network. Chord uses a hash function to map objects to nodes, and uses the hashes to efficiently search for the object. Chord lookups require a maximum of O(log N) hops to locate the destination node when node information is up to date. The performance degrades gracefully as routing information becomes less accurate.

Each node generates its own identifier by calculating a hash of its IP address. The nodes are then ordered around an identifier ring in a clockwise direction according to the value of their key. The identifiers in the ring have a value between 0 and $2^m - 1$, where m is the number of bits that the hash function generates. m should be long enough to ensure that it is unlikely for multiple nodes or multiple keys have the same hash identifier. The hash function ensures that object mappings are distributed evenly between the nodes in the network.

A node's successor is the first node that is positioned clockwise from it. Its predecessor is the first key anti-clockwise from it. Figure 1-4 shows a few nodes distributed around a chord ring where node 14 is node 7's successor and node 7 is node 59's successor. Each object is handled by the first successor node of the object's key. To allow a node to look the object up the node that owns the object can copy it to the object key's successor node. For a more efficient approach, the owner can notify the object's successor where the object resides. When the successor node receives a request for the object, it returns the address of its owner. When a node leaves the network, all of its keys are moved to its successor node.

Each node only needs to know its successor node to be able to do a lookup for a key. A node does a lookup by asking its successor to look the key up. Each successor does this until the node that controls the object is found. This is a very inefficient way to do lookups, as the number of hops to the destination node does not scale well and the maximum number of hops is n – 1 where n is the number of nodes in the network.

To improve routing performance Chord uses a list of nodes called a finger table. The finger table contains m nodes where m is the hash key length in bits. Entry i in the list contains the first node that succeeds $n + 2^{i-1}$ where n is the node's identifier. This ensures that a node has a lot of information about the nodes close to it and less information about nodes that are further away. When a node receives a lookup, it first checks whether the object's is between the node's key and its successor node key. If the key falls between the two nodes, it knows that its successor node is also the successor of the key so it can return the address of the successor. Otherwise, the node uses the finger table to forward lookups to the closest preceding node of the object's key. The nodes in the finger table are chosen to ensure that each successive lookup is forwarded at least halfway to the object's successor. As lookup gets closer to the destination node the more information a node has about other nodes near it, which increases the likelihood that it has the address of the lookup key's predecessor. Chord uses the finger table to implement a distributed binary search.

A new node generates its hash identifier and then joins the network by asking a node in the network to do a lookup for the identifier's successor node, which the joining node uses as its successor.

To ensure accurate lookups each node in the network needs to ensure that its successor information is up to date. The nodes do this by periodically doing network stabilisation where a node asks its successor's predecessor information. If that node is the same node as the node doing the stabilisation, the successor is up to date. Otherwise, it replaces its successor with the successor's predecessor. Next, it notifies the successor of its existence, this allows the successor to update its predecessor information. When a node receives a notify message it checks if the notifying node is closer than the current predecessor, if it is, or the node has no predecessor it replaces the predecessor with the notifying node, otherwise it does nothing. The stabilisation allows existing nodes to learn of new nodes in the network.

Every node calls the fix_fingers function periodically to create a finger table and to keep it up to date. Each time the function is called, it does a lookup for the next entry in the table.

To determine whether a node's predecessor is still active it regularly calls the check_predecessor function, which sends a ping message to the predecessor and listens for a reply. If the node's predecessor does not respond to the ping it removes the predecessor information to allow it to accept any node when it receives a notify message.

For correct routing, each node needs to know the correct successor node. When a node's successor fails, it needs to replace it with the failed node's successor. Because the finger table only has information about the nodes that are 2^{i-1} away it is not sufficient to determine the node's new successor. To replace the successor it needs extra information about the first few successor nodes. These nodes are stored in the successor table. When a node determines that its successor has failed, it replaces it with the next node in the successor list. The stabilise function keeps the successor list up to date by requesting its successor's successor list and remove the last node in the list and adds the successor to the front of the list. The node can also use the successor list to search for nearby nodes during lookups to improve the networks performance.

Figure 1-4 shows a Chord network consisting of six nodes that have completely up to date routing information. When a new node joins, it generates a hash identifier of its IP address. Assuming that the new node's identifier is 33 and it only node it knows that is connected to the network is node 59. Node 33 asks node 59 to do a lookup for the successor of key 33. Node 59 chooses the closest node that precedes 33 in its lookup table, which is node 14. Node 59 sends a lookup request to node 14, which in turn, consults its finger table and sends the lookup to node 24. Node 24 determines that the lookup key (33) is between it and its successor, node 40. Thus, node 40 is 33's successor. Node 24 returns a message containing the address of node 40 to node 14, which forwards the message to node 59, which then sends the result to the joining node. Node 33 then notifies node 40 of its existence. As node 33 is closer to node 40 than its current predecessor (node 24) it replaces its predecessor with 33. The next time node 24 calls its stabilise function by requesting node 40's predecessor it will receive node 33's which is closer than its current successor, so it will replace its successor. At this stage, all of the nodes in the network are routable but the networks performance will be below optimum until the new node has created its finger table and the other nodes have updated theirs. During this time if many nodes have joined

and left the network, some of the messages may hop past the destination due to the outdated finger tables. This can cause some messages to go around the network ring multiple times before reaching the correct node. One approach to avoid many messages from continuing to travel around the network is to only allow them to do a limited number of hops, if that number is exceeded, the receiving node discards the message it if it is not the destination.

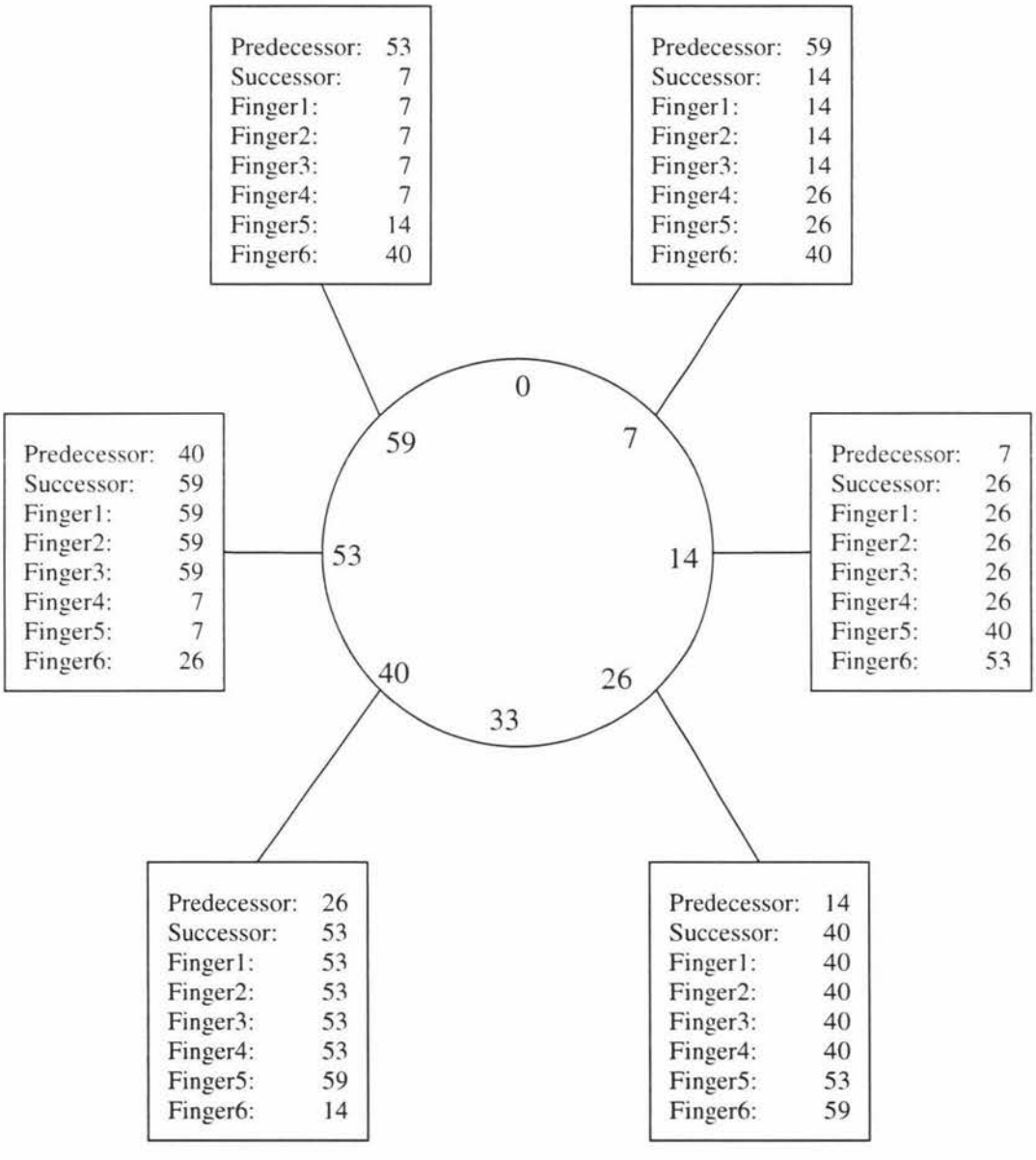

Figure 1-4 Chord routing

The Chord network allows nodes to leave the network without notifying any other nodes, but by notifying its successor and predecessor of its departure and informing the successor of the departing node's predecessor and the predecessor of the departing node's successor the changes are immediate. The nodes do not need to wait for the stabilise and notify functions to update node information when they are automatically called. The departing node can also send its keys to the successor node, which makes them immediately available instead of waiting for the objects' owners to reregister the keys.

1.2Fault tolerance and Process Migration

Checkpointing is a technique used for fault tolerance and resource reallocation. Job or process state information is stored periodically during execution to allow crashed processes to resume execution at the last saved state instead of wasting completed operations by starting from the beginning. This is especially useful for long running jobs that are impractical to restart whenever a crash occurs. It also enables the scheduler to reallocate a process to a different host where it can continue working from the saved state for load balancing or systems such as Condor [1] where resources availability may change during job execution.

1.2.1Single Process checkpointing

Condor implements user-level checkpointing [2] where a program is linked with a library that transparently implements checkpointing for the program; the program code does not contain any explicit checkpointing. Checkpointing can also be implemented at kernel-level and at user-level. In a kernel-level implementation the kernel automatically saves state information, the program does not need to be modified in any way to utilise this kind of checkpointing. In the case of application level checkpointing, the applications contain code to handle checkpointing themselves.

When a program that is linked with the Condor checkpointing library is started it installs a signal handler, creates the data structures that will contain temporary checkpoint data in the process's data segment and calls the program's main() function.

When the signal handler receives the checkpoint signal it stores all the checkpoint data to the data segment. The data segment also contains the heap, initialised, and un-initialised data. The stack is saved by simply copying its contents to the data segment; however, it cannot be restored by copying the stored stack over the new stack because the data of the procedure doing the process restoration will be overwritten. Instead, the stack is restored by swapping the stack in the restored data segment with the process stack.

The data segment of each program that uses dynamically linked libraries contains mapped segments that are links to those libraries. The links in mapped segments can be loaded in different memory addresses each time the program is started; this means that they also need to be checkpointed

When files are opened the overlay library implementation of open() is called which checkpoints the parameters and then does an open system call. The position of the file marker is saved each time a checkpoint is saved. Condor also saves the process's signal handling attributes in a table in the data segment; it also recreates pending signals by sending the signal to itself. The final type of information that is stored is the values contained in the CPU registers; this data is saved by using the signal mechanism that automatically saves and restores these values when a signal is handled.

After the signal handler has saved all the checkpoint data to the data segment it copies it to a file or socket. When the process is restarted the new data segment is overwritten with the data segment stored in the checkpoint file.

The major disadvantages of this implementation are the lack of support for checkpointing parallel or communicating processes and the need for re-linking of programs that need to implement checkpointing as the application source code is needed to do this.

1.2.2Parallel program checkpointing

Pruyne and Livny [6] discuss the implementation of consistent checkpointing where a checkpoint server is controlled by the system scheduler to store and retrieve state information of parallel message passing programs. This type of program requires that state information of each process and the state of messages buffers as well as messages already in transit are stored or that there are no more messages in transit. Codine [4] requires that all processes of a parallel job are being executed or none of them are executed to ensure that processes don't need to wait for suspended processes as they are also suspended.

With Codine parallel checkpointing, a process sends a signal to the other processes to request a checkpoint. To ensure that that all messages have been delivered the processes send a ready message to each other. When a process receives a ready message from all the other processes in the job it knows that no more messages are pending and can continue to checkpoint its own data, any messages received before the ready message are buffered. During job reconstruction, each process's data is restored in the same way as a single process job. When the job is restarted, new identifiers are created for each process. To enable the processes to use the old identifiers overlay functions map the old identifiers to the new identifiers during execution. The messages contained in the buffer are forwarded to the process the first time it calls a receive function by the overlay receive function.

1.2.3Process Highjacking

Zandy, Miller and Livny [5] implement transparent checkpointing and process migration with a technique called Process Hijacking. Process Hijacking uses dynamic program re-writing to add checkpointing to a running process. The hijacker adds checkpoint and remote procedure call capabilities to the application process that can be migrated to another host. A shadow process is then created on the original host, which handles context-sensitive calls such as file I/O for the hijacked process through remote procedure calls.

Process highjacking is highly dependent on the reliability of multiple hosts in the system it requires the availability of the original host as well as the new executing host to continue running when the process is moved. The additional overhead of doing network communication for every I/O operations can significantly reduce the performance of the program, especially across a low bandwidth high latency network.

1.3 Security

1.3.1 Grid Security Infrastructure

GSI [11] is an implementation of the Generic Security Services API (GSS-API) standard based on the Secure Sockets Layer (SSL) protocol. It provides secure communication over an open network for the Globus toolkit [8] . Resources within a computational grid are controlled by different organisations; each organisation enforces its own security policy for its own resources. Processes are allowed to act on behalf of a user. It allows security implementation across organisational boundaries and delegation of credentials.

Single sign-on and user proxy creation

Single sign-on is accomplished by mapping a global Globus user certificate to a local login name and password for each resource. The system allows dynamic allocation and de-allocation of resources that makes it impractical for a user to log into each resource individually; instead, a user proxy is used to act on behalf of the user. The user creates a user proxy credential by signing a tuple containing the user's id, name of the local host, the time the proxy will be valid as well as any other information required for authentication. The user proxy credential is then used to authenticate itself and identify its user.

Mapping registration

A resource's mapping table [13] is used to map global user names to local resource accounts. To reduce system administration workload users add their own mappings to the mapping table. The user and resource proxies authenticate each other with their global credentials, next the user proxy sends signed mapping requests containing the user's global and local name to the resource proxy, next the user logs on to the resource with local resource credentials and starts the map registration process that sends a mapping request to the resource proxy. If the request from the user process matches the request from the mapping process it proves to the resource process that the user is in possession of the local and global credentials, which means that the mapping can be added to the resource proxy.

1.4Resource allocation

Globus uses a resource proxy, which is an agent, used as an interface between grid security operations and local security mechanisms as well as scheduling jobs. The resource proxy uses a credential signed with the resource's certificate to authenticate its identity. When a new job is created resource allocation starts with the user and resource proxies authenticating each other, next the user proxy sends a resource request to the resource proxy. If the resource is available and the user is allowed to access it the resource proxy creates a process on the resource. It also creates a temporary process credential for all of the user's processes running on the resource. This allows processes to authenticate themselves.

If a process needs to use a resource not assigned during job start-up the process and the user proxy authenticate each other, then the process sends a signed request to the user proxy which then handles the request on behalf of the process by forwarding the request to the resource proxy. Allocation then proceeds in the same manner as during job creation, after which the user proxy signs the resource handler and sends it to the requesting process.

1.4.1 Distributed Dynamic Scheduling

The purpose of a scheduler is to ensure that the system load is distributed as evenly as possible and that each job receives the maximum execution time. Scheduling policy can be divided into three parts [14] :

i. The transfer policy is used to decide when a process should be transferred

ii. The selection policy determines which process needs to be transferred.

iii. The location policy dictates which node a process should be migrated to.

A physically distributed dynamic scheduler resides on every node in the system where it decides where processes should be scheduled during execution. This type of scheduler is very robust because the system can continue working if a single or a few nodes fail where a system with a centralised scheduler would fail if only a single scheduling node fails.

Implicit Scheduling

Dusseau, Arpaci, and Culler [15] propose an algorithm that uses local scheduling to implement a dynamic distributed scheduler. Executing processes are monitored to determine their communication and synchronisation behaviour, this information is then used to decide whether the process should be blocked or allowed to wait for communication without blocking. If communication is likely to conclude faster it is allowed to wait, this reduces the wasted execution time of switching processes. If the process's communication is likely to take longer to complete than a context switch, it is blocked to allow other processes to utilise the processor.

The amount of time the scheduler allows a process to wait for events before blocking is called the spin-time. The spin-time is determined by comparing the potential improved local performance of switching to another process and the global cost of processes running on remote hosts that will need to wait for the blocked process, which produces a longer job completion time.

Self-coordinated Local Scheduling

Du and Zhang propose a distributed self-coordinated scheduler [16] for heterogeneous networks of workstations. It explicitly divides the computing power of a host between user and parallel jobs; this ensures that performance and responsiveness is guaranteed for both user and parallel jobs when they are executed concurrently but are allowed to use each other's resources when they are available.

A workstation's Power Weight refers to its computing power relative to the fastest host in the network of workstations. The host with the smallest power weight will usually be the bottleneck forcing the job's other processes to wait for it to catch up wasting processor time. However, by only allowing the other processes to use the same power weight on their hosts the processes remain coordinated as well as allowing other jobs to use the rest of the allocated parallel job computing power of the faster workstations.

1.5Message-Passing computing

The Message Passing Interface and Parallel Virtual Machine libraries provide interfaces for developing parallel programs that send messages between the program's processes. A limited MPI implementation that runs on top of a Chord peer-to-peer network is discussed in section 2.5.

1.5.1Message Passing Interface

The Message Passing Interface specification [20] and [21] defines a standard for implementing a communication library for C and Fortran 77 parallel computing applications. The standard is more focused on how an MPI implementation should behave rather than defining how it should be implemented allowing the developers to implement it in a way that suits a particular architecture for maximum performance.

It provides point-to-point communication where a process communicates directly with another process, as well as collective communication that allows all of the processes in a job to communicate with each other with a single function call.

MPI processes can be divided into groups where each process in the group has a unique process identifier. Communicators allow collective calls to communicate with a subset of processes in a job. Intra-communicators are used for communication between processes that reside in the same group while inter-communicators allow processes to communicate with processes in different groups.

In the MPICH implementation [22] all of the MPI functions are implemented in terms of the Abstract Device Interface (ADI). The ADI is implemented for the specific architecture, which ensures that the minimum amount of code needs to be ported to each architecture, while ensuring good performance on every architecture.

To allow the incremental porting of MPICH to a new architecture it includes an ADI implementation in terms of the channel interface. The channel interface provides minimal architecture specific data transfer capabilities. The channel interface implements three messaging protocols:

Eager

In the eager protocol, the sending process sends the message immediately even if the receiving process is not expecting it. The receiving process stores the message in a buffer until the program calls the receive function. With this approach, the sender can send more information than the buffer can hold in which case the messages are discarded.

Rendezvous

With the rendezvous protocol the sender sends control information to the receiver, when the receiving process is ready it requests the data from the sending process. The rendezvous protocol ensures that the sender only sends as much data as the receiver can handle.

Get

In the get protocol, a process receives a message by directly reading the sending process's memory and copying it to its own memory. This provides the best performance for hardware such as shared memory systems.

1.5.2Parallel Virtual Machine

PVM [23] provides a framework for parallel programs to use a collection of heterogeneous computers as a single parallel machine. The user uses the PVM library to write programs that are divided into parallel tasks that communicate with each other to complete a job. Unlike MPI where the programmer determines the number of processes in a job before it starts, PVM programs can start new cooperating task at any time during the job's execution.

A PVM system consists of the PVM daemon that runs on every node in the system and the PVM communication library that is linked to PVM programs. The PVM daemons are responsible for routing messages to the destination, authentication, process control, and fault detection. To run a program the user starts a local PVM daemon that acts as the master. When a new task is started, the master daemon starts a PVM daemon on the host where the task will run, the new task then only communicates with its local node which in turn routes the message to the destination host. PVM also allows direct routing where a task sends a message directly to the destination task to eliminate the delay of routing through the PVM daemons. To increase security PVM daemons only communicate with other PVM daemons that have the same owner.

The local PVM daemon assigns a global unique task identifier to each PVM task that is used to task to send and receive messages from specific tasks. PVM tasks can join and groups where each task has an instance number that is unique in the group. This allows the tasks to communicate with each other in a similar way to MPI where each task's instance number is 0 to t-1 where p is the number of tasks in the group. A task

can join or leave a group at any time during execution without notifying any other tasks.

2 Implementation

This section describes the implementation of a peer-to-peer system that provides a framework for a programmer to write and run message passing interface programs. The system is divided into the communication, peer-to-peer network, job control and message passing interface layers.

Figure 2-1 shows the composition of the layers. The communication layer handles low level messaging. The peer-to-peer network level is responsible for establishing and maintaining the peer-to-peer network as well as providing a lookup service to the higher levels. The job control layer is responsible for finding the required resources to start a job, submitting the job and allows the user to kill an executing job. The message passing interface layer provides limited MPI functionality to the user program that hides the implementation of the system from the programmer. The user program can be a parallel MPI application that uses the P2P-MPI functions or a sequential program that is does not use MPI.

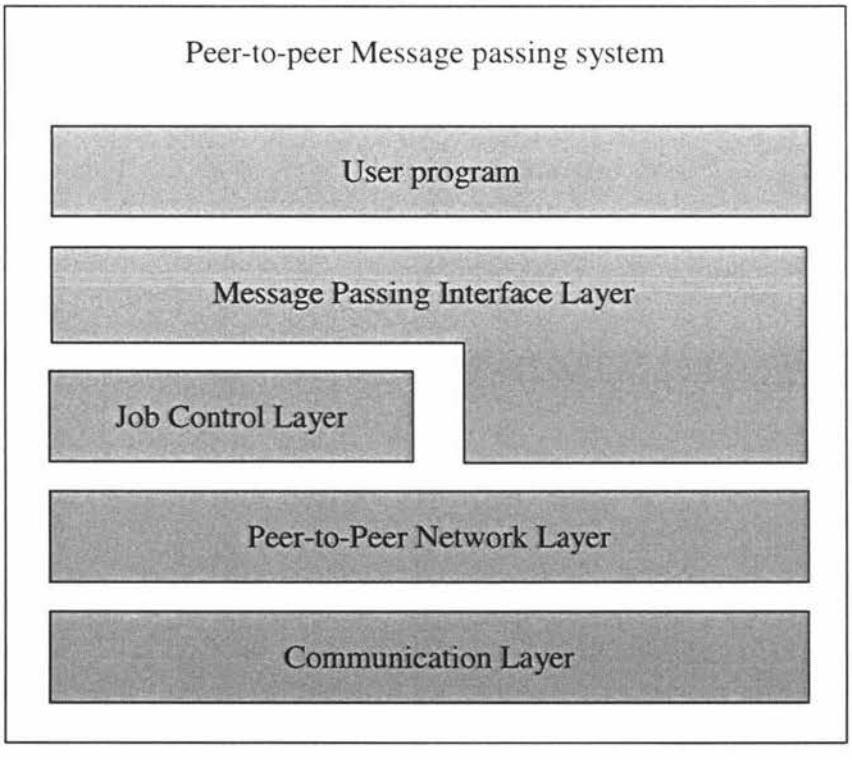

Figure 2-1 Peer-to-peer message passing system layers

2.1 Communication

The communications classes implement object oriented TCP stream sockets that simplify sending and receiving messages and files. Figure 2-2 shows the main messages sent between the communication layer classes.

2.1.1 Socket class

The socket class simplifies connecting, sending and receiving from TCP stream sockets for the programmer by accepting variable sized messages and dividing them up into standardised sized segments by using the Message and MessageSegment classes and sending the segments, the socket class then reassembles the segments into the original message at the receiving end. The socket class also provides functionality to send the contents of a file. It does this by reading blocks of the file and sending them like normal character messages, to indicate the final block of the file the sendfile function adds –1 to the end of the final block, a 0 value is added to blocks that are not the final block. The receiving socket simply writes the file contents to a file after it checks and removes the final character in the block. If the character has a value of –1, it knows it is the final block and stops receiving, otherwise if it is a null character it continues on and receives the next block.

2.1.2 MessageSegment

The MessageSegment class contains a standard number of characters that form a part of a larger message as well as the length of the data that is in the class. The length field is necessary to ensure that when the data is less than the number of characters the receiving side only reads the necessary part of the message. A negative length indicates that the segment is the last segment in the message.

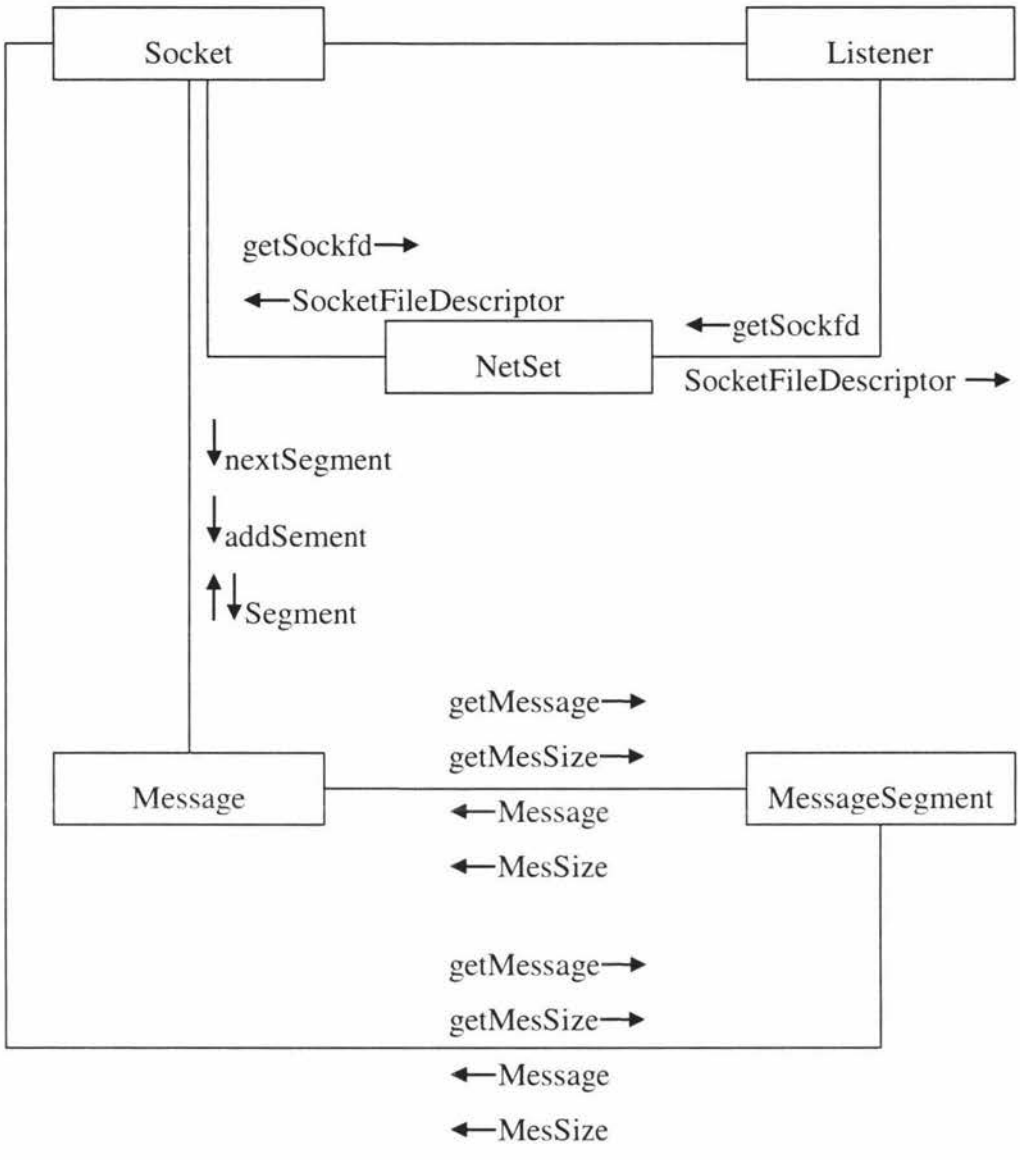

Figure 2-2 Communication collaboration diagram

2.1.3Message

The message class is responsible for dividing a message into segments contained in MessageSegment objects that the socket class can send. On the receiving end the socket class passes each message segment it receives to the message class, which then adds the segment to the message.

2.1.4Listener

The Listener class is responsible for listening for TCP connections on a specified port from Socket objects and accepting them. When it accepts a connection the accept function returns a Socket object that can be used to send and receive messages to and from the connecting host.

2.1.5NetSet

NetSet is a template class that allows synchronous multiplexing of Socket or Listener objects. A NetSet object can contain a set of either Socket objects or Listener objects and monitors the set until it at least one of the sockets or objects is ready to receive or until the timeout has expired if a maximum wait time was set. The recvReady function returns a vector of objects that are ready to receive or accept, if the timeout has expired it returns an empty vector. If the maximum time was not set it will wait until one of the objects are ready to receive. The NetSet class can be used to ensure that a node does not wait indefinitely to receive data if the sending node has failed. The NetSet is limited because it can only monitor Socket or Listener classes. This means that it cannot listen for new connections while it is waiting to receive from a socket.

2.2Peer-to-peer Network

2.2.1Content Addressable Network implementation

This section presents the implementation details of the CAN algorithm discussed in section 1.1.2. The cooperation between CAN components is shown Figure 2-3.

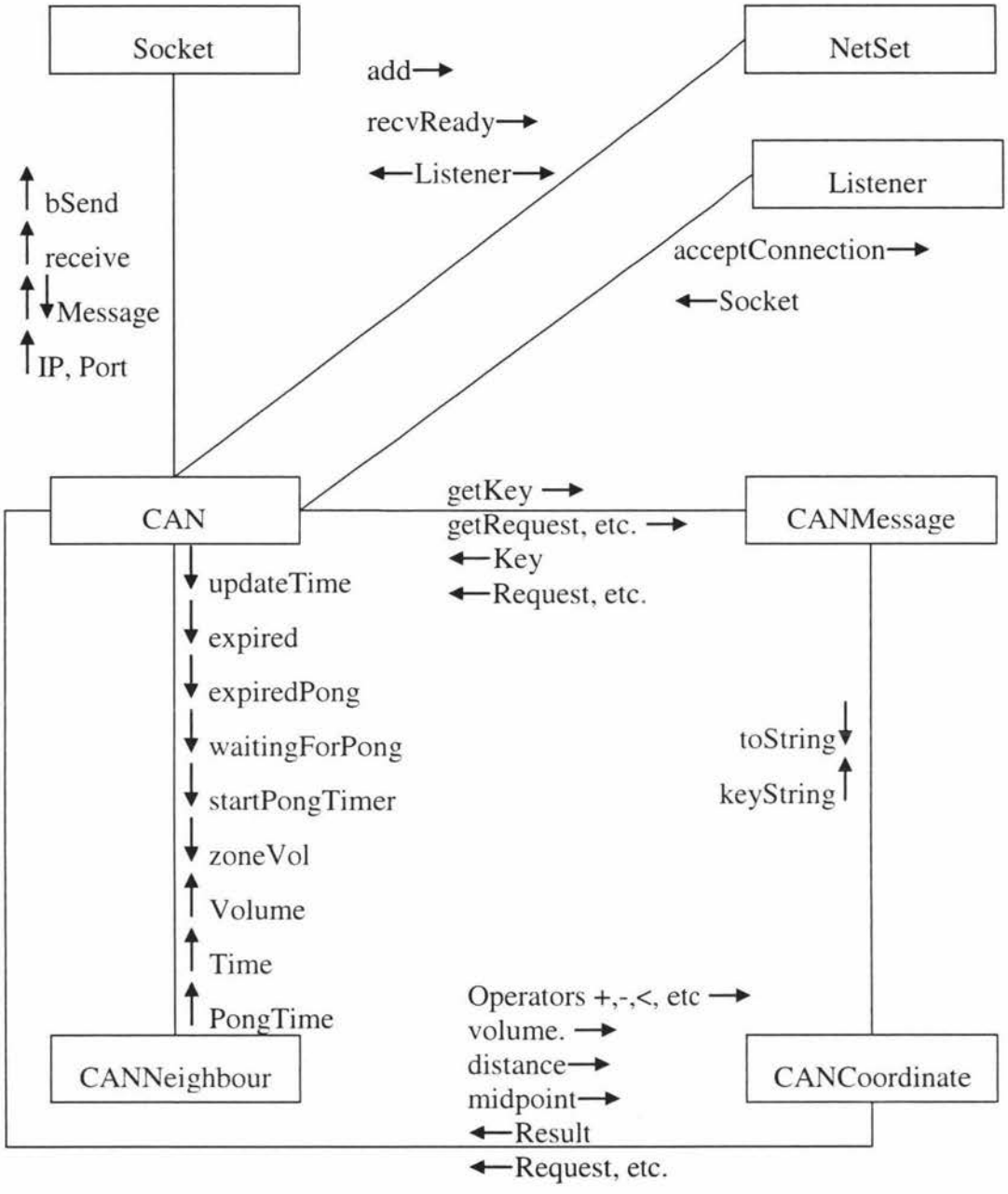

Figure 2-3 CAN Collaboration diagram

2.2.1.1 CANCoordinate

The CANCoordinate class represents a coordinate in a CAN network as a set of integers where each integer represents a position along one of the dimensions. The higher the number of dimensions in the class, the shorter the routing distance between nodes.

2.2.1.2 CANNeighbour

A CANNeighbour object handles information about one of a node's neighbouring nodes. It keeps track of the neighbour's address and zone borders as well as information about the last access time, which is the last time the node received either a PING or a PONG message from it. When the CAN class finds that the maximum amount of time since the last access time has expired it sends a PING message to the neighbour and calls the CANNeighbour's startPongTimer function to start a timer that is used to limit the amount of time the node waits for a PONG reply message. If the pong timer exceeds the limit the CAN class determines that the node has failed. If it receives the PONG message before the timer has timed out it resets the last access time and stops the pong timer.

2.2.1.3 CANMessage

The CANMessage class creates and parses text messages for transmission over the CAN network. Each message has the following format:

Key	Request	Sender IP	Sender Port	Min Zone	Max Zone	New Neighbour IP	New Neighbour Port	Message

Key

The key field contains the destination CAN coordinate of the message.

Request

The request field specifies the type of message and what the receiving node should do when it receives the message.

Network maintenance requests

JOIN

When a new node joins the network it sends a message that contains join in the request field to one of the nodes in the network that will route the message to the node that controls the zone containing the message key.

ACCEPT

If a node receives a join message from a new node it divides its zone into two equal parts and sends an accept message to joining node that contains the border coordinates of its half of the divided zone.

CHANGENEIGHBOUR

After the joining node is accepted the node that is dividing its zone sends change neighbour messages to the joining node. These messages notify the new node of the addresses and zone coordinates of its new neighbours. The dividing zone also sends change neighbour messages to the existing neighbours to notify them of changes in the neighbouring zones.

DEPART

To gracefully depart from the network a node sends a depart message to the neighbouring node with the smallest zone. If the neighbour has the same zone size as the departing node, it merges its zone with that of the departing node. If it has a different size, it cannot merge and must merge with another neighbouring node and take over the zone of the departing node. Each node sends a depart message to its smallest neighbour until two nodes of equal size are found that can merge.

PING and PONG

Each node keeps track of the last time it received a ping or a pong message from its neighbour. When the maximum time has expired, it sends a ping message and starts a pong timer. When a node receives a pong message, it replies with a pong message. If the neighbour does not reply with a pong message before the pong timer expires the neighbour has failed. If it replies before the time expires, the last access time is updated. When a node receives a ping message, it also updates the last access time of the neighbour. It is then not necessary to send a ping message before the updated timer expires.

Network testing requests

TESTDEPART

The testing program can send test depart messages to any node in the network to force it to leave the network gracefully by sending a depart message to its smallest neighbour to allow the neighbouring nodes to take over its zone. This is used to test the network performance while it is changing without failures.

TESTFAIL

To test the network's handling of node failures, the testing program sends test fail messages to nodes to force them to leave the network without notifying any other nodes in the network.

TESTROUTE

The testing program sends a test route message to a random node with a random coordinate to count the number of hops the message takes to its destination. Each node that receives a message that contains TESTROUTE in the request field sends the message back to the testing program to tell it that one hop has occurred.

TESTREPORT

When the network is being tested, each node periodically reports its zone border coordinates to the testing program.

SenderIP

The senderIP field contains the address of the original node that sent the message

SenderPort

The senderPort contains the listening port of the node that sent the message.

MinZone and MaxZone

The minimum zone field specifies the lowest coordinate of a node's zone and the maximum zone field specifies the maximum coordinate of its zone. Only messages that contain CHANGENEIGHBOUR or ACCEPT use the minimum and maximum zone fields. In an accept message the minimum and maximum zone fields contain the

values to which the joining node should change its zone. In a change neighbour message the zone fields specify the zone of the new neighbour.

NewNeighbourIP and NewNeighbourPort

The NewNeighbourIP field contains the IP address and the NewNeighbourPort contains the listening port number of a new neighbouring node. These fields are only used together with a change neighbour message.

Message

The message field contains a text message or other nested CAN messages. A node nests a change neighbour message into a depart message to allow the receiving node to handle all of the messages at one time.

2.2.1.4 CAN class

Joining

A joining node first generates a random coordinate that is contained by the network's coordinate space. As Figure 2-4 a) shows, the coordinate is then used as a key of a message that contains JOIN in the request field that is sent to any node in the network. The network then routes the key to the node that controls the zone. The receiving node then divides it zone along the dimension where the zone is the longest. If all of the dimensions have equal length, it divides the zone along the lowest dimension and changes its zone borders to contain one of the halves of the original zone. Next it sends an ACCEPT message to the joining zone that contains the borders of its new zone. In Figure 2-4 b) the accepting node then notifies the joining node of its neighbours. If the accepting node hat no neighbours it sends change neighbour messages containing itself as every neighbour and sets all of its own neighbour to point to the joining node. This allows messages to wrap around the coordinate space between the maximum and minimum coordinate in every dimension to ensure the shortest path to the destination when other nodes join in between the two nodes.

If the accepting node already has any neighbours, it sends itself as one neighbour in the dimension it has divided its zone, replaces the neighbour on the same side as the joining node with the joining node, and sends the old neighbour to the joining node.

In Figure 2-4 c) it also sends the addresses of the nodes that neighbour the accepting node to the joining node, the two nodes share common neighbours where the neighbours are bigger along the bordering axis.

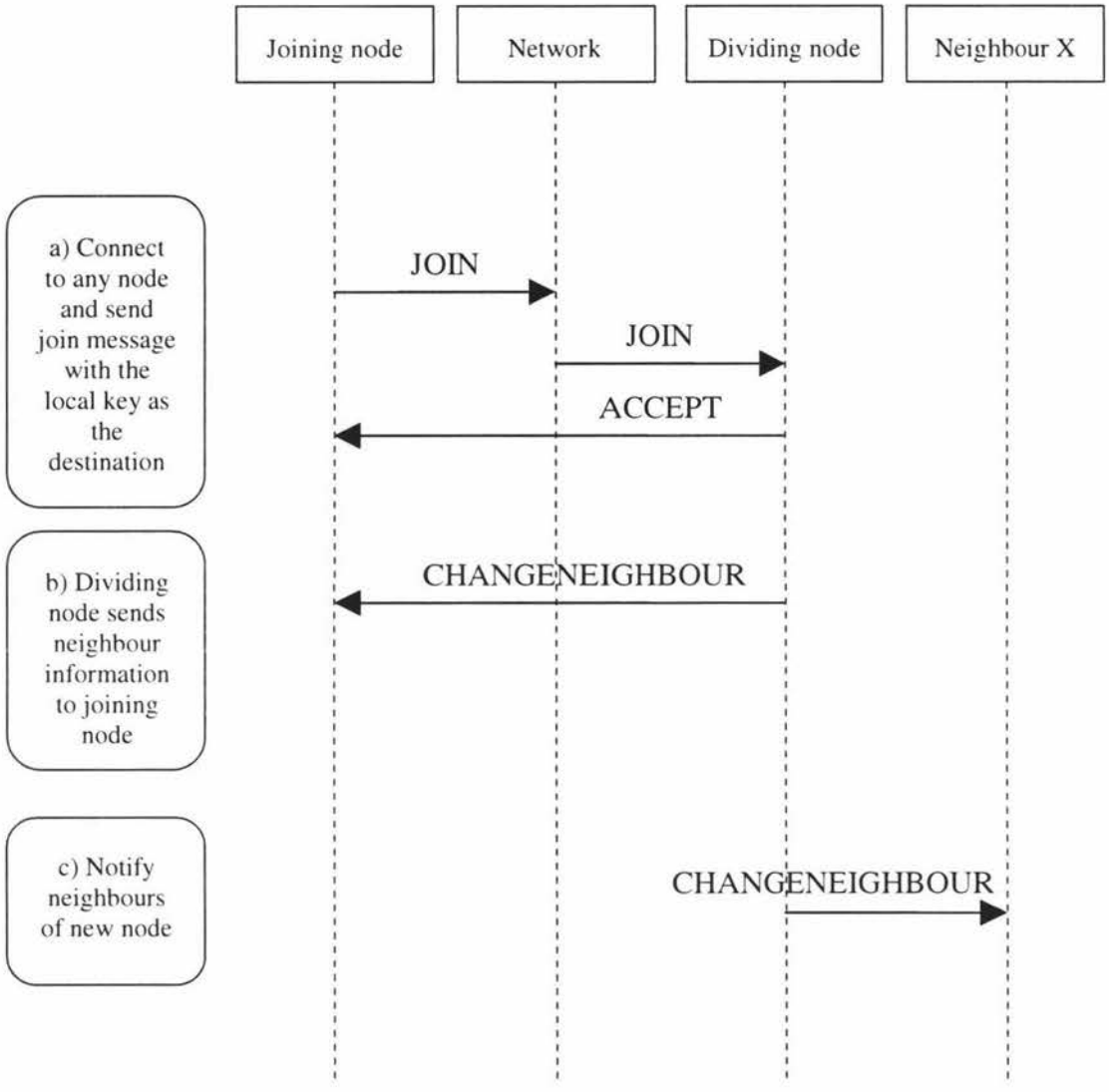

Figure 2-4 CAN Network join sequence diagram

When a node receives a change neighbour message it determines along which dimension the new neighbour shares a border with the node. It then stores the neighbour's information in the neighbour table where the row number corresponds to the bordering dimension and column corresponds to the side where the neighbouring zone lies in relation to the node. If the midpoint of the neighbour's zone is less than

the node's midpoint in the bordering dimension the neighbour is stored in position zero, if it is greater it is stored in position 1. The new node is now ready to route messages towards the destination nodes.

Routing

When a node receives a message, it first determines if it is the message's destination node by checking if the node's key is between the node's zone border coordinates, if it is it accepts the request and handles it appropriately. Otherwise, it searches through the neighbour table for the node that is the closest zone midpoint to the destination coordinate. The distance is calculated as the minimum distance of routing the message directly or by routing the message so that it wraps around the coordinate space. The distance is calculated as $x_1^2 + x_2^2 + x_3^2 + \ldots + x_i^2$ where i is the number of dimensions in the coordinate and x_i is the is smallest distance between the destination and the neighbour's zone midpoint in dimension i by either routing directly to the destination or wrapping around the edge of the coordinate space.

Departing

To leave the network a node's zone needs to be merged to form the same zone that was divided to give the node's current zone or two other nodes are needed to merge in the same way leaving one of the nodes free to take over the departing node's zone. To allow the departing node to leave the network quickly it sends a depart message to the last node with which it divided its zone or if it has not divided its own zone it sends the message to the node that divided its zone when the departing node initially joined the network. The departing node nests change neighbour message in the depart message that the node that take the zone over uses to update its neighbour nodes.

As shown in Figure 1-1 and Figure 1-2: When a node receives a depart message it first checks if it can join its zone with that of the departing node by checking if the two zones were formed by a single division of a larger zone. If the zones can merge, the node incorporates the departing node's zone into its own zone by simply updating its minimum or maximum zone attributes and notifying its neighbours of the changes. If it cannot merge the zones, it checks whether it can merge with the zone of the last node with which the zone was divided. If it is possible the node sends a depart message to that zone and takes over the zone of the departing node by changing its

minimum and maximum zone coordinates to the values contained in the depart message and updating the neighbours to the values of the change neighbour messages nested in the depart message.

If the node cannot merge with that neighbour either, it forwards the depart message to that neighbour which then tries to merge with the last neighbour with which it divided a zone. This continues until the message reaches two nodes can merge with each other, then the node that received the depart message sends its own depart message to its neighbour which then incorporates the zone in the message with its own and updates its neighbours with the change neighbour messages nested in the depart message. The other node then takes over the zone of the original departing node and changes its neighbours. Finally, the nodes that have changed their zones and neighbours notify their new neighbours of the changes in the network.

2.2.2Chord implementation

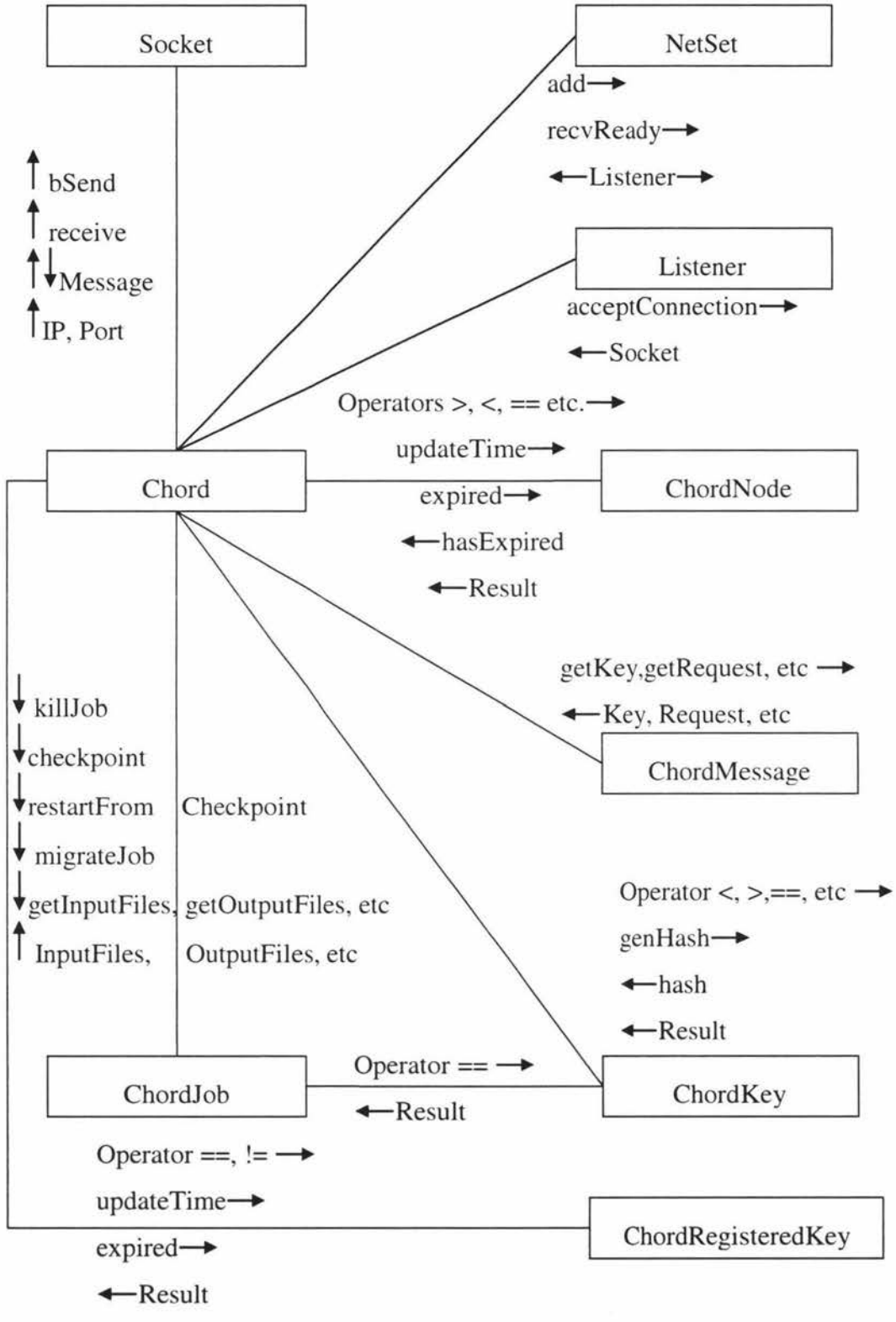

Figure 2-5 Chord collaboration diagram

2.2.2.1 ChordMessage

The ChordMessage class is responsible for creating and parsing text messages for transmission over the Chord network. Each message has the following format:

ChordKey	Request	Address	LookupAddress	TTL	Message

ChordKey

The ChordKey field contains the position in the network ring where the message should be sent.

Request

The request field defines the type of message and can contain any one of the following strings:

Network maintenance requests

NOTIFY

A node sends a notify message to its successor to tell it of its existence. This allows the successor to update its predecessor information.

GETSUCCESSOR

A GETSUCCESSOR message requests the receiving the node to do a lookup for the node that is a successor of the key in the ChordKey field.

SUCCESSOR

After a successful lookup, a node returns a successor message that contains the address of the successor's main and lookup processes.

GETPREDECESSOR

A message with a GETPREDECESSOR request indicates to the receiving node that it should return a message with the address of its immediate predecessor.

PREDECESSOR

A node returns a message that contains PREDECESSOR in the request field as well as the main and lookup process addresses of its preceding node.

PING

Each node periodically checks whether its successor and predecessor nodes are still active when its last access time expires by sending a PING message and waiting for a reply, if the node replies the node's last access time is updated. If a successor does not reply it is replaced by the next node in the successor list. If a predecessor does not reply it is replaced by an empty node which ensures that the first node sends a NOTIFY message replaces the empty node.

PONG

When a node receives a PING message from a node is it updates the last access time and returns a PONG message to show that it is still available.

Advertising and lookup requests

REGISTER

To advertise a resource on the network a node calculates a hash of the resource's name to generate a Chord key and periodically sends a REGISTER message to the successor of that key.

GETREGISTEREDKEY

After a node or the job submission tool has looked up the successor node of a resource key it can send a message that contains a GETREGISTEREDKEY request asks the receiving node to return the addresses of all of the nodes that are advertising the resource with the key and name it is searching for.

REGISTEREDKEY

If a node has keys registered with it that match the key in a GETREGISTEREDKEY message it returns a set of REGISTEREDKEY messages that contain the addresses of the nodes that are advertising those keys. When multiple registered keys match the message key, the node nests each message inside the message field of another message to form one message that contains all of the matching keys.

NOREGISTEREDKEY

If a node has no keys that match the key in a GETREGISTEREDKEY message it returns a NOREGISTEREDKEY message to indicate that no matching keys exist.

REGISTERJOB

A node that is executing a job periodically sends a REGISTERJOB message that contains its own address to the successor node of the process's ID to advertise where the process is executing.

GETREGISTEREDJOB

A job submission tool or an MPIp2p job can send a GETREGISTEREDJOB message to look up a process on the process ID's successor node.

REGISTEREDJOB

When a node receives a GETREGISTEREDJOB message and has a registered job with process ID that match the message's key, it returns a REGISTEREDJOB message that includes the address of the executing node.

NOREGISTEREDJOB

If none of a node's registered job's process IDs match, the key in a GETREGISTEREDJOB message it returns a message that contains a NOREGISTEREDJOB request.

Job control requests

SUBMITJOB

When the job submission tool has looked up and found nodes that are advertising the resource a job requires it sends a SUBMITJOB request to the node to determine if the node can accept a new job.

SUBMITCKPTJOB

A node that is leaving the network while it is running a process or receives a MIGRATE message from the submission tool needs searches for nodes that match the

job requirements and the sends a SUBMITCKPTJOB message to one of those nodes to request migrating the job to that node.

ACCEPTJOB

If a node receives a SUBMITCKPTJOB or a SUBMITJOB, message and it can run the process without exceeding its maximum allowed number of jobs it replies with an ACCEPTJOB message.

REJECTJOB

When a node receives a SUBMITCKPTJOB or a SUBMITJOB message and the maximum number of allowed jobs are already running on the node, it sends a REJECTJOB message back.

KILLJOB

The job submission tool can kill a running process by sending a KILLJOB message that contains the process's ID to the executing node.

KILLEDJOB

If a node is executing the process with the same ID as the key in a KILLJOB message it the process and replies with a message containing KILLEDJOB in the request field.

NOSUCHJOB

If a node receives a KILLJOB or MIGRATEJOB message from a job submission tool with a Chord key that does not match the Ids of any of its running processes it notifies the submission tool that the process does not exist with a NOSUCHJOB message.

MIGRATEJOB

The job submission tool can force a node to migrate an executing process to another node by sending a message containing MIGRATEJOB in the request field. This type of message is used to test job migrations.

MIGRATEDJOB

After the successful completion of migrating the process to a different node the previous node sends a MIGRATEDJOB message back to the job submission tool.

CHECKPOINTJOB

The CHECKPOINTJOB request is used for testing. When the job submission tool sends a checkpoint job request to a node the node makes a checkpoint of the running process. If the CKPT_CONTINUE environment variable has any value the process continues after the checkpoint, otherwise it exits.

CHECKPOINTEDJOB

After a node has completed a forced checkpoint, it returns a checkpointed job messages to the job submission tool.

Network testing requests

TESTREPORT

If the testing variable in the TestConfig.h file is set to true every node will periodically send a TESTREPORT message to the testing program to update it with the node's current successor and predecessor node IDs.

TESTFAIL

The testing program sends a TESTFAIL message to a node to force it to exit immediately without notifying any other nodes in the network. This is useful to determine how well the network handles random failures.

TESTDEPART

The testing program can send a TESTDEPART message to a node to tell it to leave the network gracefully by notifying the necessary nodes of its departure. This is used to test how the network handles node departures.

TESTLOOKUP

The testing program periodically sends TESTLOOKUP messages to random nodes with random keys to count the number of hops it take to find the key's successor node. When a node receives a test lookup message it sends a copy of the message to the test node and then continues the lookup like a normal lookup with the exception that TESTLOOKUP is used in the request field instead of GETSUCCESSOR. This is used to calculate the average number of hops a message takes.

Address

The address field contains the Internet Protocol address and port of the main process of the node that sent the message or the address of the node that was located by a GETSUCCESSOR, GETPREDECESSOR, TESTLOOKUP, or GETREGISTERED-JOB request.

LookupAddress

The lookup address field contains the same IP address as the address field but the port number of lookup process instead of the main process of the node.

TimeToLive

The time to live field contains the maximum number of hops between nodes a lookup message has left. The TTL value is decremented at each node that does the lookup. If the TTL reaches zero the receiving node discards the message and returns an empty message to signify that the correct successor was not found. This is used to prevent a message from repeatedly looping through the network and wasting bandwidth when the finger table is in an incorrect state, when the network has recovered the message can be sent again.

Message

The message field can contain any text or binary data. Some messages such as REGISTEREDKEY message use the message field to add other nested REGISTEREDKEY messages to send them as a single message.

2.2.2.2 ChordKey

The ChordKey class represents a position in the Chord network ring as an n-bit integer with a maximum value of 2^n-1. The class can calculate the integer value by applying a hash function to a string.

2.2.2.3 ChordNode

The ChordNode class handles information about remote nodes that are used for routing in the Chord network. This information includes the node's ID, main process

address and lookup process address as well as the last access time. Every time the node receives a message from a predecessor or successor node it calls the node's updateTime() function. It also allows the node to check periodically whether the node is still active, if the timeout has expired, it can send a ping message and listen for a reply.

2.2.2.4 ChordRegKey

The ChordRegKey class stores temporary information about a key that maps to a process or a resource as well as the addresses of the node that is advertising the object. Unless the node that owns the object updates the information regularly, the information's lifespan is limited to prevent old object advertisements from presenting outdated and inaccurate information when nodes fail or leave the network.

2.2.2.5 Chord class

The Chord class implements the Chord peer-to-peer object location algorithm.

Joining the network

A node joins the network by first creating a node ID by calculating a hash from its IP address and listening port. This hash ID determines the node's position in the network ring. Next, the node process forks to create another process. The new process starts listening for messages while the original process goes through a list of previously known nodes and tries to connect to any one of them. When it finds a node in the network, it requests it to do a lookup for the successor of its hash ID by sending a message containing GETSUCCESSOR in the request field and the hash ID in the key field. When the existing node returns the address of its successor, the connecting node sets the returned node as its successor. The node the contacts its new successor by sending a notify message to allow the successor to make the new node its predecessor if it is closer than its current predecessor is. After the node is connected, the main process starts building the finger and successor tables by asking its successor to do the initial lookups.

For testing purposes, all the nodes connect to the first node started in the network; this simplification was used to ensure that nodes that are not part of the network do not connect to each other and create a separate network from the existing network. In a large network that is permanently available nodes can prevent this from happening by first determining if the node is connected to the network by checking if its successor has the same key as the node. The node can determine this by doing a lookup of the key that immediately follows the node's ID. An extension to the Chord protocol can be implemented where the node simply sends a message to ask whether the node is connected to the network. This would require that the first node to join the network be able to detect that no other nodes exist and then create the network by telling connecting nodes that it is connected to the network even though it is the only node in the network.

Departing from the network

To gracefully leave the network the main process of a departing node notifies the main processes of its successor and predecessor nodes that it is leaving the network and pass its predecessors information on to the its successor and its successor's information to its predecessor. To ensure that keys are not temporarily lost the node can transfer all of its registered keys and process Ids to its successor.

Handling failed nodes

When a node fails or leaves the network without notifying any nodes its successor and predecessor will detect the failure when the node's timeout expires and it does not reply to a ping message or when sending it a message fails. After the main process detects its successor's failure, it replaces it with the first node in the successor table. If there are no nodes in the successor list it replaces the successor with the first entry in the finger table that does not have the same address as the current successor. If the successor and finger tables do not contain any nodes, it cannot replace the successor. The node must then join the network as if it were a new node in the network. If any processes are running on the node they continue, but MPI communication may be interrupted if the processes need to do lookups to locate other processes.

If the main process detects the failure of its predecessor, it replaces it with an empty node that acts as a placeholder. The next time the main process receives a notify message it replaces the predecessor placeholder with the sender of the message.

Processes

Due to the implementation of the messaging part of the system where lookups work like remote procedure calls in that the system waits for the result for a lookup until it either receives a result in a return message a process would block until it receives a result or the lookup times out. Thus, a single process would not be able to service requests from other nodes while waiting for a lookup. By using two permanent processes as well as extra processes to handle requests, the system can continue working without blocking other node's requests.

Main process

The main process of a node is responsible for keeping routing information up to date. It only handles messages such as notify, that inform the node of the state of the network and do not require a returned message. All other message are handled by the lookup process. The main process maintains the finger and successor tables by doing lookup requests for the successor nodes of specific location in the network and sends the contents of to the finger table through a pipe to the lookup process. The main process also advertises the node's resources on the network.

Lookup process

The lookup process provides a lookup service for the local submission tool and other nodes on the network. It uses the information it reads from the receiving end of the pipe to keep its finger table up to date. When it receives a lookup request the lookup process forks, which creates a process to handle the request. The lookup process then continues to listen for other lookup requests while the newly created process completes the lookup returns the results and exits.

The lookup process also accepts and executes job processes for the node. It maintains a list of processes running in the network of which it is the successor node. While a job is running, it advertises the process's ID on the network.

Handling requests

When the lookup process receives a lookup request, it immediately forks to create a new process to handle the request.

2.2.2.6 Resource advertising

A node advertises its resources on the network. It calculates a hash key of the resource name and periodically sends a register message to the successor node of the key. The register message contains the key and the resource name. The destination adds the resource to its registered key list and starts a timer. Each time it receives a register message for a key that is already in the key list it resets the registered key's timer. If the key's timer exceeds its maximum time, the node removes the key from the list. This ensures that the resources a node has advertised are removed if it leaves the network or fails. The resource advertising technique is flexible enough to allow a node to advertise any string such as the node's architecture, processor speed and available memory. The nodes can also advertise other information such as the node's geographic location that would allow the user to specify that a program run on nodes in one city to reduce communication latency.

2.2.3XML Job description file (XMLJobDoc)

To submit a job to the system the user must provide an XML p2pjob document that specifies all the information necessary to submit and move a job as well as retrieving the results of the job.

Each Job file can contain the following tags; the submission tool and nodes ignore any other tags:

The <executable> element is a mandatory field that contains the name of the executable file.

The <arguments> element is an optional field that contains a list of the command line arguments that are passed to the executable at execution.

<inputfile> is an optional field used to specify any files that need to be submitted along with the executable file.

<outputfile> is an optional element used to specify any files that are created by the program during execution that should be returned to the user when the job has completed. These files are also submitted to the new executing node along with the checkpoint and input files during migration.

At least one <requirement> field is required. The submission tool uses this field to match nodes on the network that are advertising these resources.

The <processes> element is a required field used to specify the number of required nodes.

```
<?xml version="1.0"?>
<p2pjob>

<executable>p2pApp</executable>
<arguments>arg1 arg2</arguments>

<inputfile>input1</inputfile>
<inputfile>input2</inputfile>

<outputfile>outputfile</outputfile>

<processes>2</processes>

<requirement>Linux</requirement>
<requirement>x86</requirement>

</p2pjob>
```

2.2.4Job Submission tool

In Figure 2-6 a), the submission tool parses the job's xml file to obtain the value of the requirements field in the job description XML file and generates a SHA1 hash of that value. Next in Figure 2-6 b), it sends a message to the local node and that contains GETSUCCESSOR in the request field and the requirement hash in the key field. The local node then looks up the successor node of the key value and returns its address to the submission tool.

The submission tool then sends a GETREGISTEREDKEY message that contains the requirement string to the successor of the key to request the values that match the hashed requirements as shown in Figure 2-6 c). In reply, the successor node sends a list of the nodes that are advertising the resources specified in the requirement string. If the successor node returns a NOSUCHKEY message it means that, no nodes are currently advertising the required resources. If there are nodes available or there are fewer nodes that have the necessary requirements specified in the XML job file the job submission fails.

However if the job's requirements are met the submission tool calculates the process keys for each of the processes by hashing a concatenated string of the submission tool's local IP address and the process number in the set. Next in Figure 2-6 d), it sends a SUBMITJOB request each of the nodes in the list. If the node replies with a REJECTJOB message, the submission fails. If the node replies with an ACCEPTJOB message, the submission tool first sends the XML job file followed by the input files (Figure 2-6 e). Then it sends a key list of all of the processes running in the job. Finally, it transfers the executable file to the executing node.

The submit tool can also kill any processes running in the job by doing a Chord lookup for the process key to locate the executing node and sending a KILLJOB request to that node. Similarly, it can send a request to a node migrate a process to a different node, this is only useful for testing purposes.

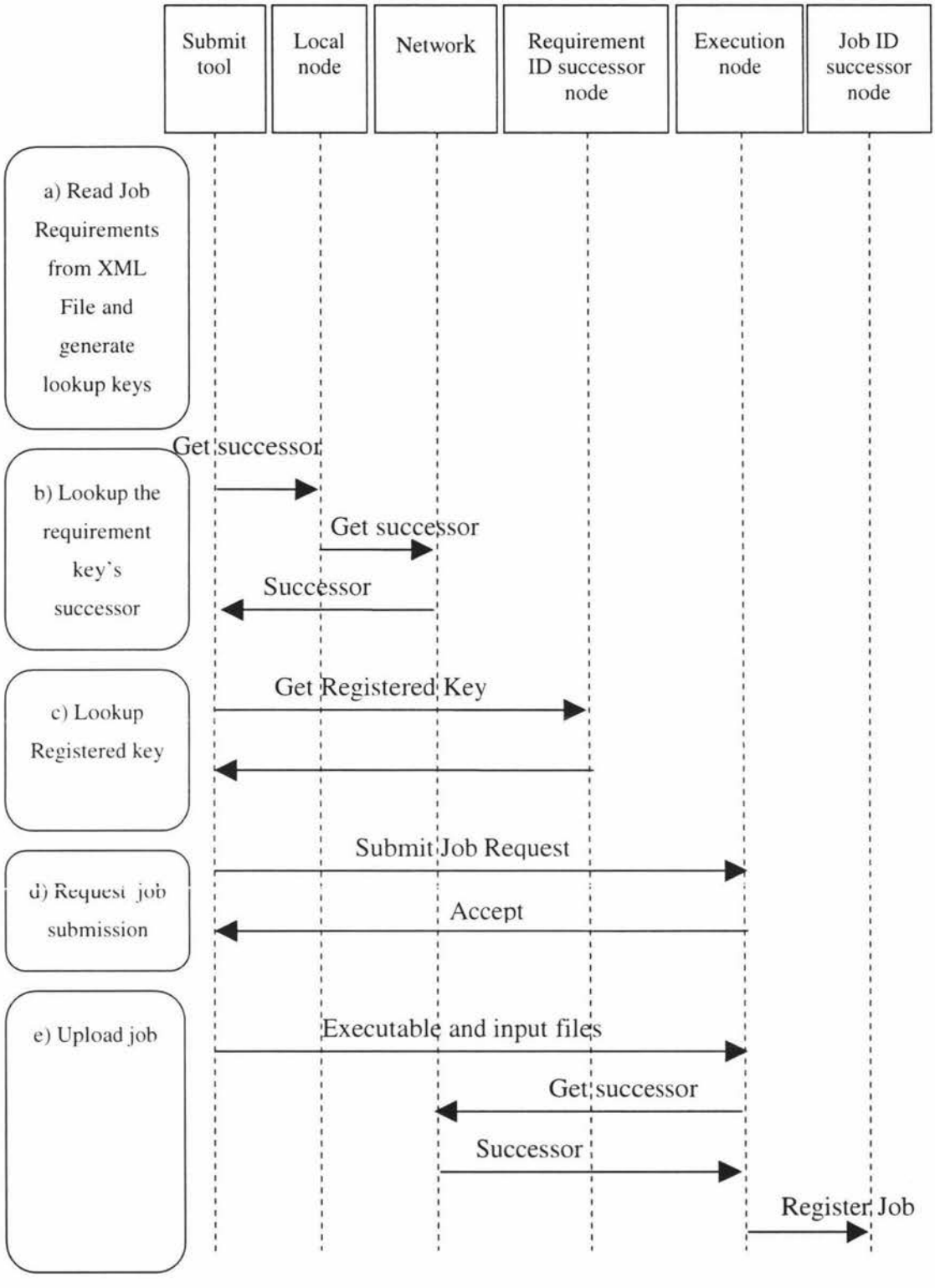

Figure 2-6 Job Submission sequence diagram

2.3Chord Job execution

When a node receives a job submission request, it checks whether it is already running the maximum number of processes if the resources are available, it accepts the job otherwise it rejects it. On accepting a job, the node creates a temporary directory that will contain all of the files that are associated with that job during its existence. Next, the node receives the job XML file and parses the contents to determine the names of the input files it will receive as well as the number of processes that are running in the job. Next it forks, the original process returns to continue listening for messages. The new process continues to receive the input files, the keys of al the processes in the job and the executable. It then uses the process keys to determine the rank of the processes and sets the MPI_RANK environment variable to that value. Finally, it starts the job process.

During job execution, each executing node periodically checks if its job process is still running, if it is it periodically advertises the job's key on the network by registering the process key on the successor node of the process key to allow the other processes in the job as well as the job's owner to locate the process. Each node limits the number of jobs it will execute at a time, while the maximum number of jobs are executing on the node it will reject any new job request.

2.4Job migration

When a node needs to leave the network it needs to move all of the executing processes to a different node before it can leave. The first step in migrating a process is to send the checkpoint signal to the executing process, this activates the checkpoint function of the ckpt library that was developed by the Condor[1] team. The CKPT_FILENAME environment variable contains the name of the file that the ckpt library creates to store the contents of the process. While the ckpt library is checkpointing the process the node does a lookup to find another node that satisfies the process requirements defined in the XML job file. If it finds a node advertising those resources it sends it a submit checkpoint job request. If the remote node accepts

the job the executing node begins to transfer the XML job file, the input files, any output files that the program may have created as specified in the job file as well as the checkpoint file. The executing node does not transfer the original executable that the submission tool submitted as all of the information necessary to continue running is contained in the checkpoint file.

2.4.1 Files

To allow the system to migrate a process that uses files or sockets the program needs to be linked with the ckpt library to allow the programmer to use the ckpt API functions.

The ckpt API provides the following functions that allow the programmer to register functions that are called before and after a checkpoint as well as after the program restarts respectively:

```
void ckpt_on_preckpt(void (*f)(void *), void *arg);
void ckpt_on_postckpt(void (*f)(void *), void *arg);
void ckpt_on_restart(void (*f)(void *), void *arg);
```

The ckpt library does not checkpoint open file handlers. To checkpoint file information, the application programmer must implement it. The following code segment shows how checkpointing files may be implemented in an application by using the checkpoint library's API. The preckpt function is registered by calling the ckpt_on_preckpt function and the onrestart function is registered by calling the ckpt_on_restart function. When the program receives the checkpoint signal the signal handler calls preckpt, if the file is open preckpt flushes the file buffer to ensure that buffered data is not lost, then it saves the position of the file's position indicator and closes the file.

After the job transfer and restart has completed and before the program continues from the last point of execution, the onrestart is called. The onrestart function reopens the file and moves the file position indicator to the original position.

```
int offset;
```

```
int openfile;

void preckpt(FILE* file){
     if(file != NULL){
          openfile = 1;
          flush(file);
          offset = ftell(file); //get file offset
          fclose(file);
     }
}

void onrestart(FILE* file){;
     if(openfile){
          file = fopen(filename,"r");
          fseek(file, offset,0); //move to saved offset
     }
}
```

This method is only useful for migrating processes, to allow processes to roll back to an old checkpoint the state of the file will need to be saved each time the process is checkpointed.

2.5 Peer-to-peer Message Passing Interface

The system provides a minimal MPI API implementation that is compatible with the MPI standard from the programmer's point of view

The following MPI functions are implemented:

```
int MPI_Init (int *argc, char ***argv);
int MPI_Finalize ();

int MPI_Comm_rank (MPI_Comm comm, int *rank);
```

```
int MPI_Comm_size (MPI_Comm comm, int *size);
double MPI_Wtime ();

int MPI_Send (void *buf, int count, MPI_Datatype
datatype, int dest, int tag, MPI_Comm comm);

int MPI_Recv (void *buf, int count, MPI_Datatype
datatype, int source, int tag, MPI_Comm comm, MPI_Status
*status);

int MPI_Get_processor_name (char *name, int *resultlen);
```

`MPI_Init` is the first MPI function an MPI program should call. It registers a function with the `ckpt_on_preckpt` that closes any open sockets when the program receives a checkpoint signal. It also reads the MPI_COMM_RANK, MPI_COMM_SIZE, and MPI_LOCALNODE environment variables created by the local executing node as well as the set of MPI_PROCESS environment variables to get a list of the Chord keys of all the processes running in the job. The init function also creates a list that will contain a socket for each of the other processes in the job.

`MPI_Finalize` is the last function a program calls; it closes all of the open sockets before the program exits.

The `MPI_Comm_rank` and `MPI_Comm_size` functions return the rank and size values that the `MPI_Init` function read from the environment variables. The `MPI_Wtime` returns the current time, which the program can use to calculate the wall time.

`MPI_Send` first checks if a socket to the receiving process is available, if it is it sends the data and returns on success. If a connection is not available it does a lookup for the process with the process's Chord key until it finds the node, after which it connects to the process, sends its own MPI rank followed by the data. If the send fails it closes the connection and looks the process up again, reconnects, and resends the

data. This method allows processes to locate and reconnect to each other when a node leaves the network and moves its processes to different nodes.

`MPI_Receive` determines whether a connection to the sending process exists and receives data from the socket if it does. If it is not connected, it creates it listens for a connection, when a process connects the listening; process receives the rank of the sending process. If the rank is the same as the rank the function is trying to receive from it finishes receiving and returns. If the connecting process has a different rank than what the `MPI_Receive` function is attempting to receive from it saves the socket file descriptor in the list of process sockets and continues listening for new connections until the process it wants to receive from connects.

2.6 Testing framework

2.6.1 Routing testing

The testing framework allows the testing program to start a number of nodes to test either CAN or Chord network's performance in exactly the same way with the same class interface. The interface allows the program to test the networking algorithm's routing during normal situations when the network is stable as well as when nodes fail or leave the network or join the network. The program periodically starts nodes that listen on different ports on one workstation. It also regularly forces nodes either to depart gracefully or to simulate a failure by exiting. The program does not simulate the speed of a normal network connection.

2.6.1.1 Test class

The test class defines a number of pure virtual functions that its sub classes must implement to specialize the testing for the particular peer-to-peer network. It also reads the testing configuration file to determine the user specified settings. The user can specify the maximum number of nodes the program can start, the minimum port that the nodes may use to listen for incoming messages and most importantly the

ratios for starting, failing and departing nodes as well as the routing ratio. The test program uses the ratio to determine how often each action occurs by generating a random number for each action, if the random number is less than the ratio the action is executed.

2.6.1.2 CANTest class

The CANTest class implements the virtual functions defined in the testing class. It allows the test program to fail or depart specific nodes by sending a CAN message containing the appropriate request. It sends the message directly to the node instead of routing it through the system to ensure that the node executes the request as soon as possible. It also listens for message from the nodes that contain PONG, TESTREPORT or TESTROUTE in the request field. Each time it receives a test route message it increments the number of hops counter and divides the counter by the number of messages it has sent to calculate the average number of hops between nodes. If a node does not reply to a ping before the maximum time has exceeded, the CANTest object removes the node from the list of known nodes. Every time the CANTest program receives a test report message it updates the node's minimum and maximum zone borders and updates the node's last access time if the is already in the list of known node, otherwise it adds the node and its details to the list.

2.6.1.3 ChordTest class

The ChordTest class implements the Chord specific function for controlling nodes on the network and testing routing. To test the routing the testing program sends a message that contains TESTLOOKUP in the request an a random value in the key field to a random node. When a node receives the message it sends its own TESTLOOKUP message to the appropriate node and a copy of the test lookup message it received to the testing program. The ChordTest object counts these messages in the same way as the CANTest to calculate the average number of routing hops. The test program stores information about the live nodes in list of ChordNeighbour objects and uses these objects to keep track of live nodes and sends PING messages when the last access time in the ChordNeighbour objects expire. If the nodes do not reply before the NetSet recvReady times out it removes the node

from the list. Each node in the network sends information about itself to the testing program. The information includes the node's successor and predecessor node.

2.6.1.4 Test program

The test program tests the routing performance by counting number of hops a message takes to a random location in the network by sending it from a random node. The program reads a testing configuration file to determine on which port number it should listen, the nodes also read the file to determine the address of the testing program. For the tests completed in section 3.1 the cases where the test program randomly selected the same node for the destination of the message as the node it picked to do the search were ignored. The program allows the user to manually start nodes, and stop or fail nodes, as well as do routing tests in interactive mode. It can also do non-interactive testing where it automatically starts and stops nodes, and do routing tests.

During non-interactive testing, the program is in one of the following three states:

Ramping up

The program starts in the ramp up state where it creates the network and starts more nodes than it fails and departs.

Running

During the running stage, the test program starts and stops an equal number of nodes to simulate normal activity in the network.

Ramping down

During ramp down the program fails and departs more nodes than it starts.

2.6.2MPI Performance testing applications

The two applications used to compare the performance of MPICH and P2P-MPI have nearly identical source code for the different test systems, the only difference between

the source code used to test MPI on a normal cluster and the peer-to-peer MPI was the inclusion of the MPIp2p.h header file instead of the standard mpi.h header for the P2P-MPI applications. Appendix B contains the source code of the two MPI test applications. The rest of the code was left unchanged to ensure an accurate comparison.

The first MPI benchmarking application does a simple ring test to test the message passing speed of the network without any significant processing overhead. Process zero sends a message to process one, which in turn sends the message to the following process. The process with the highest identifier sends the message back to node zero to complete a loop.

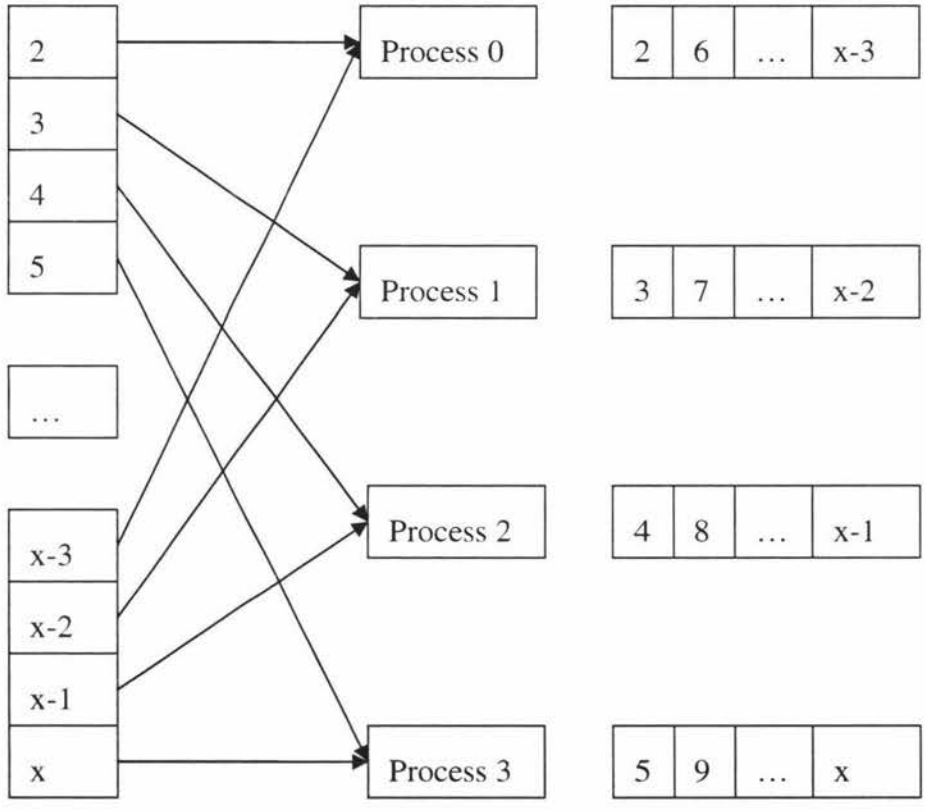

Figure 2-7 Test Factor Distribution

The second MPI testing application implements a real world application that does parallel factoring of RSA encryption keys of variable lengths where each process is responsible for testing its own set of keys in the.

Each MPI process starts by initialising its test factor variable to MPI ID + 2. Then it iterates through a loop in which it checks if the key is divisible by the test factor. If it is not, it increments the test factor by the number of processes and continues the loop until it finds one of the key's factors, or the test factor exceeds the key's square root, which is the maximum possible value of a factor. Figure 2-7 shows how the test factors are distributed among the processes. This ensures that each process tests unique numbers for divisibility. Each of the slave processes periodically sends a message to tell the master whether it found a factor; the master process then sends messages to all of the slaves to tell them if one of the processes found a factor. If the master indicates that one of the processes found a factor all of the processes exit. Two versions of the RSA program were tested, an embarrassingly parallel version and a high communication load version. Each of the processes in the embarrassingly parallel version communicates once with the master process for every ten thousand factors it tests while the high communication load processes communicate with the master once every five hundred possible factors it tests. The two versions are used to determine how different communication patterns influence the execution time of MPICH and P2P-MPI applications.

3 Results

In this section, the performance data obtained from the MPI testing programs and the peer-to-peer testing framework are discussed. The first part of this section compares the performance of the Chord and Content Addressable Network peer-to-peer networks and the second part discusses the comparison of MPI applications running on the P2P-MPI system and MPICH.

3.1 Peer-to-peer routing performance

The following tests were done to determine the routing performance of the CAN and Chord algorithms. Each algorithm was tested with different numbers of nodes when the network is in a stable state with no nodes joining or leaving the network during the tests and the routing information of each node is complete and up to date. To simplify the start-up of a large number of nodes the tests were completed on a single workstation. This does not influence the results as the test is only concerned with the number of hops between nodes it takes to locate the lookup node with Chord and the number of hops needed to route a message to the destination node with CAN.

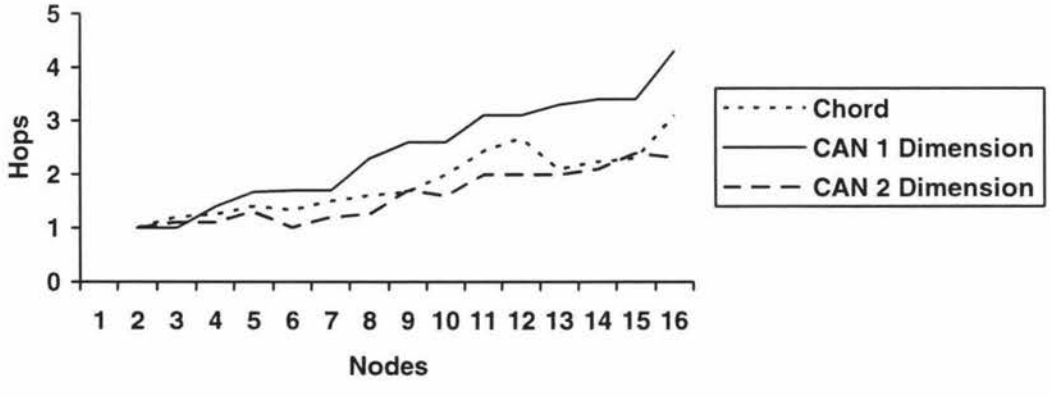

Figure 3-1 Chord/CAN routing comparison

In Figure 3-1 the growth of the number of hops required to route a Chord message as the number of nodes increase is close to O(log N) that Stoica *et al.* [17] claim. However, the number hops will increase during if many nodes join and leave the network in a short time due to inaccurate routing tables.

The number of dimensions that a CAN network uses has a big influence on the network routing performance as the network grows. The average number of hops required to do a lookup is very close to $O(n^{1/d})$ claimed by Ratnasamy, Francis, Handley and Karp [10] .

A network that utilises the CAN algorithm stabilises much faster than Chord after many new nodes join. This happens because a new node receives all the routing information it needs as soon as it joins the network whereas a Chord node needs to do a number of lookups to build a list of routing nodes before it can route lookups at its maximum efficiency.

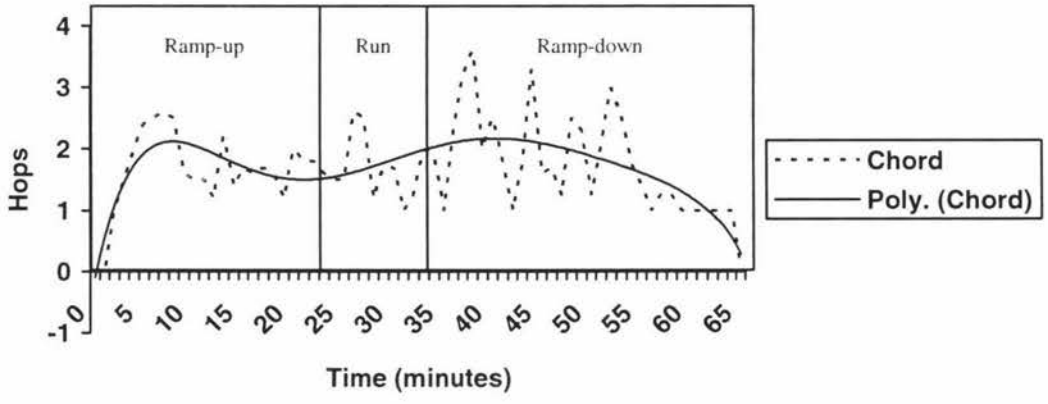

Figure 3-2 Chord routing during ramp-up, running and ramp-down states.

Figure 3-2 shows the number of hops messages require when a Chord network is ramping up, running and ramping down. During the ramp-up stage, the number of hops grows quickly as new nodes join, as soon as the nodes' routing tables start getting more up to date the required hops gradually decrease. During the running stage the hops steadily decrease as the routing tables become more complete as shown by the Chord graph. The polynomial trend line shows that the more lookups

are required when nodes start leaving the network because the some of the nodes have incorrect routing table entries. When the even more nodes leave and the routing tables are updated to reflect that the number of hops needed slowly decreases until there are no nodes left in the network.

3.2Message passing performance

The MPI tests were performed on eight nodes of a Linux Beowulf cluster. The nodes communicate to each other over a 1Gbit/s Ethernet network.

3.2.1Embarrassingly parallel RSA factoring

The embarrassingly parallel RSA application sends a message from each of the slave nodes to the master node to inform it whether is has found a key or not. The master then notifies the slaves whether any of the nodes have found a key. The embarrassingly parallel version does this once for every 10000 keys a node tests.

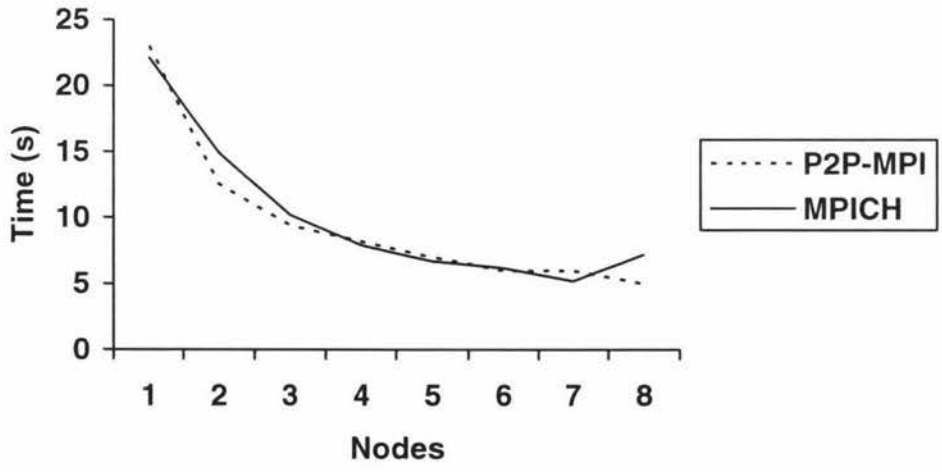

Figure 3-3 MPI Embarrassingly Parallel 56-bit key

Figure 3-3 shows the completion time of the embarrassing parallel RSA program with a 56-bit key. MPICH and P2P-MPI complete the tests in similar times. With the short 56-bit key, this shows that the initial lookup and connection time of P2P-MPI in a network with eight nodes is not large enough to affect the overall execution time of the job.

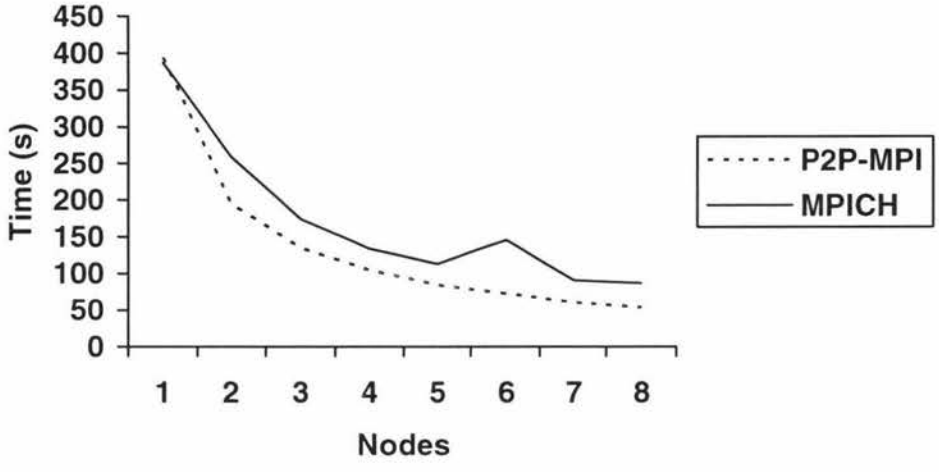

Figure 3-4 MPI Embarrassingly Parallel 64-bit key

In Figure 3-4 the 64-bit key tests show an average of 26% faster completion time for the P2P-MPI test. With a single process, the programs complete in similar times. However, with multiple nodes that require communication MPICH lags behind.

3.2.2 High communication load RSA factoring

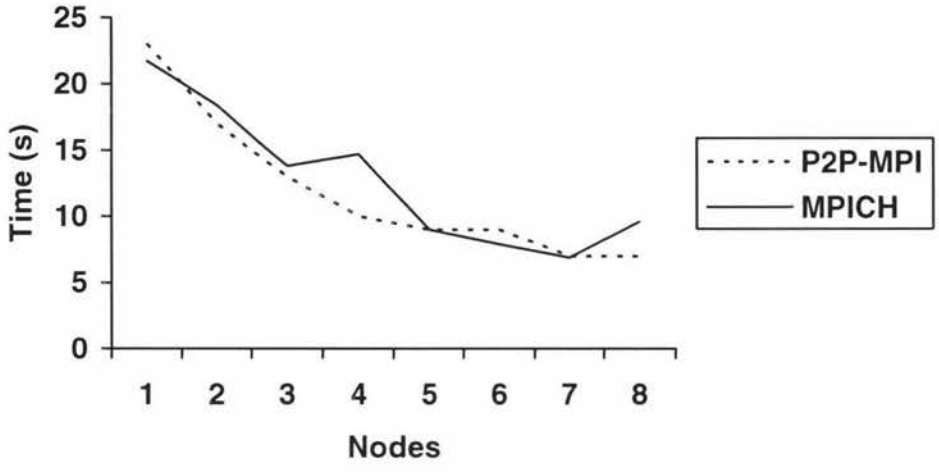

Figure 3-5 MPI Performance High communication 56-bit key

Like Figure 3-3, Figure 3-5 shows similar completion times with P2P-MPI and MPICH with the high communication load RSA application. The 56-bit key used is not large enough to show a great difference in communication time.

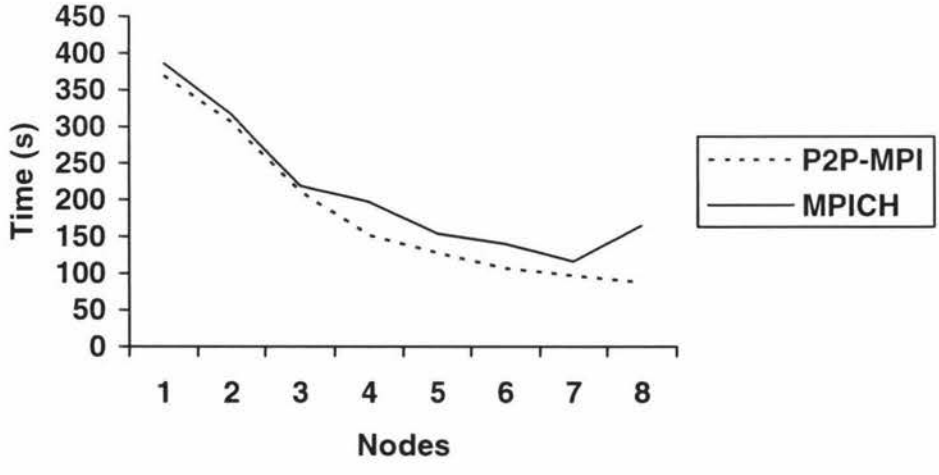

Figure 3-6 MPI Performance High communication 64-bit key

In Figure 3-6 P2P-MPI's again shows a performance advantage over MPICH with a 64-bit key. This time MPI-P2P finds the factors in 17% less time than MPICH.

The shorter completion time of the P2P-MPI benchmarks is most likely caused by the way send and receive is implemented in P2P-MPI. In P2P-MPI, a process creates a connection the first time it wants to send a message to another process and keeps it alive until it fails or the program exits, whereas an MPICH process connects to the destination every time it sends a message and uses the connection only once.

3.2.3Ring test

The following ring tests were performed to compare the systems' initialisation and communication performance without the load of a real world application.

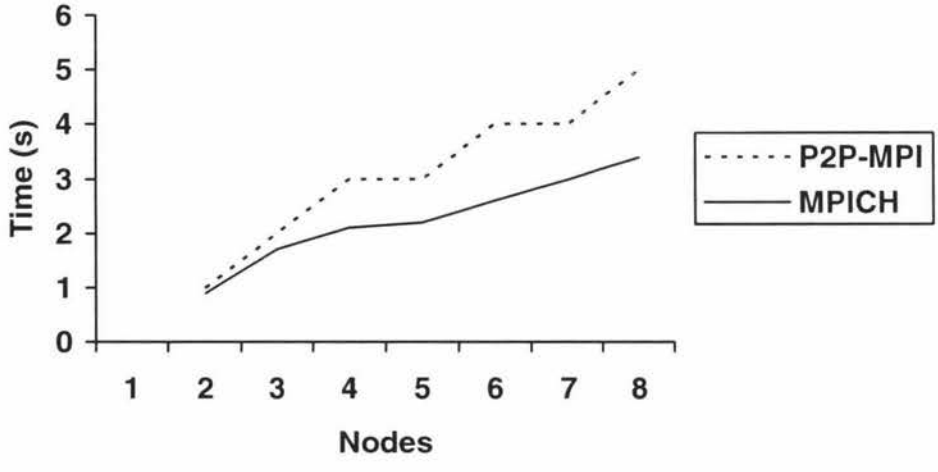

Figure 3-7 Ring Test 5000 loops

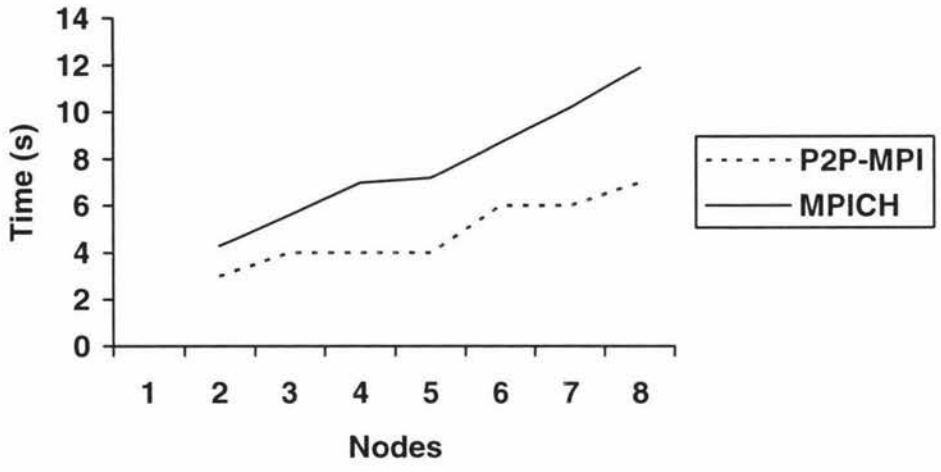

Figure 3-8 Ring Test 20000 loops

In Figure 3-7 the ring test was completed with a message looping 5000 times around a ring of nodes. During this test, P2P-MPI takes much longer to finish the test but in Figure 3-8, P2P-MPI completes the ring test much faster than MPICH with 20000 loops. These two ring test comparisons show that initialisation takes longer on P2P-MPI and is long enough to reduce the performance of jobs that that take a short time to complete. Figure 3-8 shows that the faster communication of P2P-MPI more than compensates for the slower initialisation with its faster persistent connection communication if the application needs to communicate many times during execution. However, with more nodes the lookup time will increase as the number of hops needed to route between nodes increases. The connection time for an MPICH job on the other hand stays constant with more nodes if the network's architecture does not need to change to accommodate more nodes. This means that the overall performance of each process of jobs that run on P2P-MPI for a short time will degrade with more nodes in the network.

3.3Impact of process migration

The time it takes to migrate a process to a different node is highly dependent on the amount of memory a program uses. The more memory the program uses the larger

the checkpoint file will be. A very simple test application that increments an integer in a loop, produces a 1.5MB checkpoint file. It takes 9.5 seconds for process migration from the time the migration request is sent to the time the process continues execution on the new node on a 10Mbit/s network. Compressing the checkpoint file can reduce the size of the checkpoint file to reduce its transfer time. The gzip utility compressed the checkpoint file by 61%.

In some cases, it may take longer for process migration than the job's execution. It may be faster to resubmit jobs that use large amounts of memory or run for a short time on a slow network than migrating one of the job's processes, as the executable is usually much smaller than the checkpoint file.

4 Further work

4.1 Job replication for fault tolerance

To allow processes to continue execution when processing nodes fail the system would require a mechanism for storing the state of the job at certain interval By storing checkpoints of the processes on nodes that are not executing any process will allow the job to continue working from the last checkpointed state. A new node can use the checkpoint files to restart the process from the last checkpointed state while the other processes roll back to the last checkpoint to ensure that all the processes in the job remain synchronised. To reduce the likelihood of the loss of a process's state when multiple failures occur, the checkpoint files should be distributed among nodes that are not executing the job. The system should allow nodes to upload only the sections in the checkpoint file that have changed since the previous checkpoint similar to GNU diff and patch programs to minimise the amount of data that is sent with each checkpoint.

4.2 Distributed checkpoint storage

OceanStore

Kubiawitz *et al.* [19] designed OceanStore to replicate and distribute data objects across many peer-to-peer nodes in a global network that can handle large-scale failures in the network without losing stored objects. This makes it ideally suited to storing checkpoint files during a job's execution. The checkpoint storage nodes periodically check if the executing processes are still available. If one of the checkpoint storage nodes detects a failure, it forces the other nodes in the job to download their latest checkpoint files and restart the processes. Next, it searches for a node that can provide the necessary resources to run the failed process after which it

sends a message to that node to tell it to download the checkpoint file and restart the process.

4.3 Security

A system such as this requires two types of security. First, the nodes in the system need protection from malicious user programs. Second, the user program should be able to prevent the node's users or other programs from accessing its data. The second requirement is much harder to implement securely. The user can encrypt all of the files the program needs but the programs need to decrypt the data to use it. This means that the decryption key has to be available at the node, which could make the encryption useless. Another approach an attacker can take is to modify the user's program in some way to divulge the contents of the encrypted data or use a program's checkpoint files to look at the memory contents to find unencrypted data or the decryption key.

4.3.1 User privileges

A simple and effective way to ensure that user programs cannot damage the system is to create a separate user for the peer-to-peer node. The user should have very limited privileges. It should not be allowed to execute any programs other than user jobs and should not have access to any files outside of its own directory. To prevent processes from accessing each other's data the system can create a new user for each process that only allows the process to access its own directory. This would allow the node to limit the amount of disk space a process can use to prevent a malicious program from filling the hard disk up in a denial of service attack.

4.3.2Node Access control

An owner of a node in the network may want to make it available to only a specific user or group of users. This can be achieved by using public/private key encryption. The user creates a public and private key and sends the public key to the node's owner. The owner then adds the key to a list of users that may use the node. When the user wants to use the node, the submission tool creates request message, signs the message with its private key and sends it to the node. The node checks whether the sender is in the list of allowed users, if it is not it rejects the job. Otherwise, it verifies the message signature by decrypting it with the node's public key. If the node's identity is confirmed and it has the available resources, it accepts the job. If it cannot verify the node's identity, it rejects the job request.

4.3.3Encrypted communication MPI

To prevent eavesdropping of MPI messages on a public networ the peer-to-peer MPI implementation can be extended by adding encryption facilities that are completely transparent to the user. When a job starts and initialises MPI, the master node's MPI_Init function creates a public/private and sends the public key to the other processes in the job, next the master creates a symmetric session key, encrypts it with its private key and sends it to the other nodes. This allows the processes to encrypt their messages with the symmetric key without the large overhead of public/private key encryption, which would reduce the job's performance if it requires a lot of communication.

4.3.4Job submission

The submission tool currently only searches for nodes that satisfy the first requirement it encounters in the XML job file. To handle multiple requirements the tool should find the nodes that satisfy each of the requirements and cross-reference these nodes to determine which nodes meet all of the requirements. If one of the

nodes rejects a submit request the job submission fails, the submission tool does not attempt to send submit the job to other nodes even if enough nodes are available that meet the requirements

5 Conclusion

5.1 Communication

The system does not allow any node to connect from behind a network address translation router. Each node needs to know the IP address other nodes need to use to send messages to it. If it is behind a router, its private network address is different from the address that other nodes outside the private network need to use. By getting the address of the sender from the socket's file descriptor the Socket class can determine the correct address of the sending node. The receiving node can then insert the correct address into the message instead of the sending node inserting the private address.

5.2 Job submission and migration

Job submission and migration can be simplified and made more efficient by moving all of the contents of the job directory into a single archive file and compressing it. This would reduce the amount of data to transfer and allow the node to transfer any temporary files created by the program that are not specified in the job file.

Can change the ChordJob.cpp to use a standard checkpoint file name for each job, the name does not need to be unique because each job is in a unique directory. Remove path from the directory name instead change to that directory before starting the process, this will not work if the program changes directory during execution, the checkpoint file will be saved in the current working directory.

The MPI_LOCALNODE environment variable used to store the address of the local node will not change when the process is migrated. This means that the process will still try to connect to the old address; this is not a problem if the address is 127.0.0.1 and the node runs on the same port as the previous node. However if nodes run on

different ports and new node changes the environment variable it has no effect on the environment variable value in the program. One way to allow the process to determine the new IP address and port is by storing the address in a configuration file. When the process restarts, a function registered with `ckpt_on_restart` ckpt API function opens the configuration file and reads the new local node ports.

5.3 Job execution

The system only allows a node to run one process at a time due to the implementation of the peer-to-peer MPI API. Each process in a job listens on a port determined by its MPI process identifier, this means that if two or more process from different jobs run on a node the processes can have the same MPI identifier and would try to listen on the same port. The process that started first, will run as normal, but the other process will not be able to receive message from the other processes in its job.

This can be solved by using a simple server on each node that listens for connections on a standard port number. Whenever a node creates a listener socket, it sends a message to the server that contains its peer-to-peer job identifier, MPI process number and listen port number. When an MPI process tries to connect to another MPI process it does a normal peer-to-peer lookup to find the executing node and then looks the MPI process's port number up from the port server. This will increase the time it takes to connect to an MPI process slightly but should not degrade the overall performance significantly, as the connections are kept alive as long as both of the communicating processes are not migrated. This also allows the process to choose a listen port and if it is not available, it can use any other available port. If a node process migrates to another node it does not need to listen on the same port number to ensure that the other processes find it.

Another potential problem of the implementation is the use of semi-permanent connections between processes that are created the first time two processes start sending messages to each other. If a job requires very large number of processes where every process needs to communicate with every other process, each process would require a large number of open sockets that the operating system may not be

able to provide. A very simple solution is for the sending process to connect to the receiving process, send its message and disconnect immediately. To reduce the connection time each process should store the location of another process after its initial lookup and use that address every time it needs to send a message. If the sender cannot connect to the receiving node, it will need to do a new lookup to determine the process' new IP address. To use this method effectively the user application programmer will need to group small messages together to minimise the delay of reconnecting for every message.

5.4 Summary

Modern peer-to-peer networks can provide fast and efficient lookup of resources with minimal network information. Peer-to-peer message passing parallel computing is very competitive with the Message Passing Interface standard when used with long running applications that do not require regular communication that minimises the initial overhead of finding the necessary resources as well as the time it takes processes to locate each other.

References

[1] M. J. Litzkow, M. Livny, and M. W. Mutka, "Condor: A hunter of idle workstations", *Proceedings of the 8th International Conference on Distributed Computing Systems,* pp. 104-111, June 1988.

[2] M. Litzkow T. Tannenbaum, J. Basney, and M. Livny, "Checkpoint and migration of UNIX processes in the Condor distributed processing system", *University of Wisconsin-Madison Computer Sciences Technical Report #1346,* April 1997.

[3] G. Stellner and J. Pruyne, "Resource management and checkpointing for PVM", *Proceedings of the 2nd European Users Group Meeting,* pp. 131-136, September 1995.

[4] G. Stellner, "Consistent checkpoints of PVM applications", *Proceedings of the the 1st European PVM Users Group Meeting,* 1994.

[5] V. Zandy, B. Miller and M. Livny, "Process hijacking", *Proceedings of the 8th IEEE Symposium on High Performance Distributed Computing,* pp. 177-184, 1999

[6] J. Pruyne and M. Livny, "Managing checkpoints for parallel programs", *Job Scheduling Strategies for Parallel Processing,* vol.1162, pp 140-154, 1996.

[7] K. Czajkowski, I Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith and S. Tuecke, "A Resource management architecture for metacomputing systems". *Lecture Notes in Computer Science,* vol. 1459, 1998.

[8] The Globus Toolkit Administration Guide – http://globus.org/gt2.4/admin/

[9] T. Basney, M. Livney, P. Mazzanti, "Harnessing the capacity of computational grids for high energy physics", *Conference on Computing in High Energy and Nuclear Physics,* 2000.

[10] S. Ratnasamy, P.Francis, M. Handley, R. Karp & S. Shenker, "A scalable content-addressable network", *Proceedings of the 2001 conference on applications, technologies, architecture, and protocols for computer communications*, pp 161-172, 2001.

[11] Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems", *in Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms, November 2001.*

[12] Foster, C. Kesselman, G. Tsudic and S. Tuecke, "A security architecture for computational grids", *(ACM) Conference on Computer and Communications Security*, pp 83-92, 1998.

[13] S. Tuecke, "Grid security infrastructure (GSI) roadmap", Internet Draft, February 2001.

[14] X. Evers, "A Literature study on scheduling in distributed systems", October 1992.

[15] Dusseau, R. Arpaci, and D. Culler, "Effective distributed scheduling of parallel workloads", *Proceedings of Sigmetrics International Conference on Measurement and Modelling of Computer Systems (ACM)*, 1996.

[16] X. Du and X. Zhang, "Coordinating parallel processes on networks of workstations", *Journal of Parallel Computing*, pp. 125-135, 1997.

[17] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, H Balakrishman, "Chord: A scalable peer-to-peer lookup protocol for internet applications", *IEEE/ACM Transactions on Networking,* Vol. 11, No. 1, pp. 17-32, February 2003.

[18] B. Wilkinson and M. Allen, *Parallel Programming: Techniques and Applications using Networked Workstations and Parallel Computers.* New Jersey: Prentice-Hall, 1999.

[19] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czarek, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. "Oceanstore: An architecture for global-scale persistent storage", *Proceedings of ASPLOS 2000,* Cambridge, Massachusetts, November 2000.

[20] The Message Passing Interface Forum, "MPI: A Message-Passing Interface Standard", Version 1.1, June 1995.

[21] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. "MPI: The Complete Reference" MIT Press, 1996.

[22] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum, "A high-performance, portable implementation of the MPI message passing interface standard."

[23] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. "PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Networked Parallel Computing", MIT Press, 1994.

Appendix A

Nodes	1	2	3	4	5	6	7	8
Chord	0	1	1.2	1.25	1.4	1.33	1.5	1.6
CAN 1 Dim	0	1	1	1.4	1.67	1.7	1.7	2.3
CAN 2 Dim	0	1	1.1	1.1	1.4	1	1.2	1.25

Nodes	9	10	11	12	13	14	15	16
Chord	1.67	2	2.44	2.67	2.1	2.24	2.3	3.1
CAN 1 Dim	2.6	2.6	3.1	3.1	3.3	3.4	3.4	4.3
CAN 2 Dim	1.7	1.6	2	2	2	2.1	2.4	2.32

Table 1 Data of Figure 3-1 Chord/CAN routing comparison

Time	1	2	3	4	5	6	7	8	9	10	11	12	13
Chord	0	1	1.5	2	2.4	2.5	2.6	2.5	1.6	1.5	1.5	1.2	2.2

Time	14	15	16	17	18	19	20	21	22	23	24	25	26
Chord	1.4	1.7	1.6	1.7	1.6	1.2	2	1.8	1.8	1.6	1.5	1.5	2.6

Time	27	28	29	30	31	32	33	34	35	36	37	38	39
Chord	2.5	1.2	1.7	1.7	1	1.3	2	1.9	1	2.3	3.2	3.6	2

Time	40	41	42	43	44	45	46	47	48	49	50	51	52
Chord	2.5	1.8	1	1.8	3.3	1.6	1.7	1.2	2.5	2.3	1.3	2	3

Time	53	54	55	56	57	58	59	60	61	62	63	64	65
Chord	2.5	1.8	1.4	1	1.3	1.3	1	1	1	1	1	1	0

Table 2 Data of Figure 3-2 Chord routing during ramp-up, running and ramp-down states.

Nodes	1	2	3	4	5	6	7	8
P2P-MPI	23	12.5	9.4	8.2	7	6	6	5
MPICH	22.1	14.9	10.2	7.9	6.7	6.2	5.2	7.2

Table 3 Data of Figure 3-3 MPI Embarrassingly Parallel 56-bit

Nodes	1	2	3	4	5	6	7	8
P2P-MPI	394	195	135	105	85	73	61	54
MPICH	387	259	174	134	113	146	91	87

Table 4 Data of Figure 3-4 MPI Embarrassingly Parallel 64-bit

Nodes	1	2	3	4	5	6	7	8
P2P-MPI	23	17	13	10	9	9	7	7
MPICH	21.7	18.4	13.8	14.7	9	7.9	6.9	9.6

Table 5 Data of Figure 3-5 MPI Performance High communication 56-bit

Nodes	1	2	3	4	5	6	7	8
P2P-MPI	369	306	211	152	128	107	97	88
MPICH	386	317	219	197	154	140	116	165

Table 6 Data of Figure 3-6 MPI Performance High communication 64-bit

Nodes	1	2	3	4	5	6	7	8
P2P-MPI	-	1	2	3	3	4	4	5
MPICH	-	0.9	1.7	2.1	2.2	2.6	3	3.4

Table 7 Data of Figure 3-7 Ring Test 5000 loops

Nodes	1	2	3	4	5	6	7	8
P2P-MPI	-	3	4	4	4	6	6	7
MPICH	-	4.3	5.6	7	7.2	8.7	10.2	11.9

Table 8 Data of Figure 3-8 Ring Test 20000 loops

Appendix B

Ring test program

```
#include <stdlib.h>
#include <stdio.h>
#include "MPIp2p.h"

#define loops 5000

int main(int argc, char *argv[])
{
  FILE* output = stderr;
  int myid, numproc;
  char buf[10];

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &myid);
  MPI_Comm_size(MPI_COMM_WORLD, &numproc);


  memset(buf,10,sizeof(char));

  if(myid == 0)
  {
    MPI_Status stat;
    double starttime, endtime;
    int i;
    starttime = MPI_Wtime();
    sprintf(buf,"PING");

    for(i = 0;i < loops ;i++)
```

```
    {
      MPI_Send(buf,10, MPI_CHAR, 1,0, MPI_COMM_WORLD);
      MPI_Recv(buf,10, MPI_CHAR, numproc - 1, 0,
MPI_COMM_WORLD, &stat);
    }
    endtime = MPI_Wtime();
    fprintf(output,"Wall time : %f\n",endtime-starttime);
  }

  else{
    MPI_Status stat;
    int i;

    for(i = 0;i < loops ;i++)
    {
      MPI_Recv(buf,10, MPI_CHAR, myid - 1, 0,
MPI_COMM_WORLD, &stat);
      MPI_Send(buf,10, MPI_CHAR, (myid + 1)%numproc,0,
MPI_COMM_WORLD);
    }
  }
  MPI_Finalize();
  return 0;
}
```

RSA Factoring program

```
#include <stdio.h>
#include <stdlib.h>
#include <gmp.h>
#include <time.h>
#include "MPIp2p.h"
#include "rsa.h"
#define size 100
```

```
#define debug 1

mpz_t p,q,n,c,d,e;
char out[size];
int numproc, myid;
MPI_Status Stat;
FILE * output;

void crackPQ(mpz_t returnP,mpz_t returnQ,mpz_t E,mpz_t
N){

  int counter=0;
  int found,ofound=0,flag,compare,end;

  mpz_t D,M,C,tempMes,tempN,max;

  mpz_init_set_ui(returnP,2+myid);
  mpz_init_set(returnQ,N);
  mpz_init(C);
  mpz_init(D);
  mpz_init(tempMes);
  mpz_init(tempN);
  mpz_init(max);

  mpz_sqrt (max, n);
  compare = mpz_cmp(returnP,max);
  found = 0;
  end = 0;
  while((compare<=0)){ //check if testP<=max

    if(mpz_divisible_p (N, returnP)!=0){//found factor
      found = 1;
      mpz_divexact (returnQ, N, returnP);//calculate
other factor

      mpz_get_str (out,10,returnP);
```

```
      if(debug)
        fprintf(output,"myid = %d  cracked P :
%s\n",myid,out);
      mpz_get_str (out,10,returnQ);
      if(debug)
        fprintf(output,"myid = %d  cracked Q :
%s\n",myid,out);

      mpz_add_ui(returnP,returnP,numproc);
      compare = mpz_cmp(returnP,max);
    }

    else{ //no factor found yet
      mpz_get_str (out,10,returnP);
      if(debug > 1)
        fprintf(output,"P = %s\n",out);
      mpz_add_ui(returnP,returnP,numproc);

      compare = mpz_cmp(returnP,max);
      if(compare>0){
        if(debug > 0)
                fprintf(output,"end reached myid
%d\n",myid);
        end = 1;//return;
      }
    }
    counter++;
      //counter >500 for high communication
    if((counter > 1000)||end || found){
      counter = 0;
      if(myid == 0){
        int i,f;
        ofound = 0;
        for(i = 1; i < numproc; i++){
          int of;
```

```
          MPI_Recv(&of, 1, MPI_INT, i,0, MPI_COMM_WORLD,
&Stat);

          if(of){
            ofound = 1;
            if(debug)
                fprintf(output,"Slave found factor\n");
          }
        }
        f = found || ofound;
        for(i = 1; i < numproc; i++){
          MPI_Send(&f, 1, MPI_INT, i, 0, MPI_COMM_WORLD);

        }
        if(found&&debug)
          fprintf(output,"Master found factor\n");
        if(f)
          break;
      }
      else{
        MPI_Send(&found, 1, MPI_INT, 0, 0,
MPI_COMM_WORLD);
        MPI_Recv(&ofound, 1, MPI_INT, 0,0,
MPI_COMM_WORLD, &Stat);

        if(ofound || found || end)
          break;
      }
    }
  }
}


int main(int argc, char *argv[]){

  double starttime,endtime;
```

```
  mpz_t crackedP,crackedQ;

  //Initialise MPI
  MPI_Init(&argc,&argv);

  MPI_Comm_size(MPI_COMM_WORLD,&numproc);
  MPI_Comm_rank(MPI_COMM_WORLD, &myid);
  starttime = MPI_Wtime();

  output = stderr;

  mpz_init_set_si(e,79);
  mpz_init_set_str(n,argv[1],10);
  mpz_init(d);
  mpz_init(c);
  mpz_init(crackedP);
  mpz_init(crackedQ);

  crackPQ(crackedP,crackedQ,e,n);

  if (myid ==0){
      endtime = MPI_Wtime();
      fprintf(output,"The wall time was : %f
seconds\n",endtime-starttime);
  }

  MPI_Finalize();
  return 0;
}
```