Copyright is owned by the Author of the thesis. Permission is given for a copy to be downloaded by an individual for the purpose of research and private study only. The thesis may not be reproduced elsewhere without the permission of the Author.

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

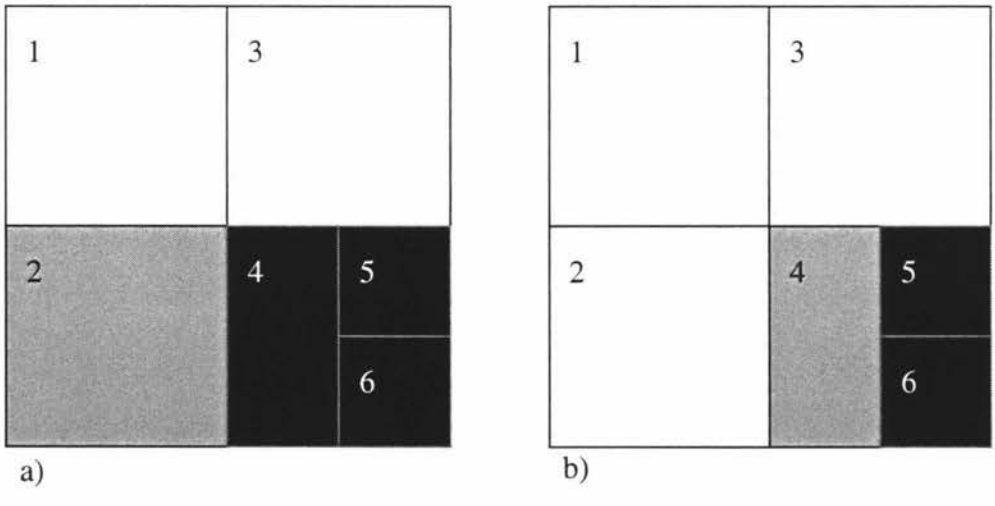

Figure 1-1 Merging of CAN zones during departure

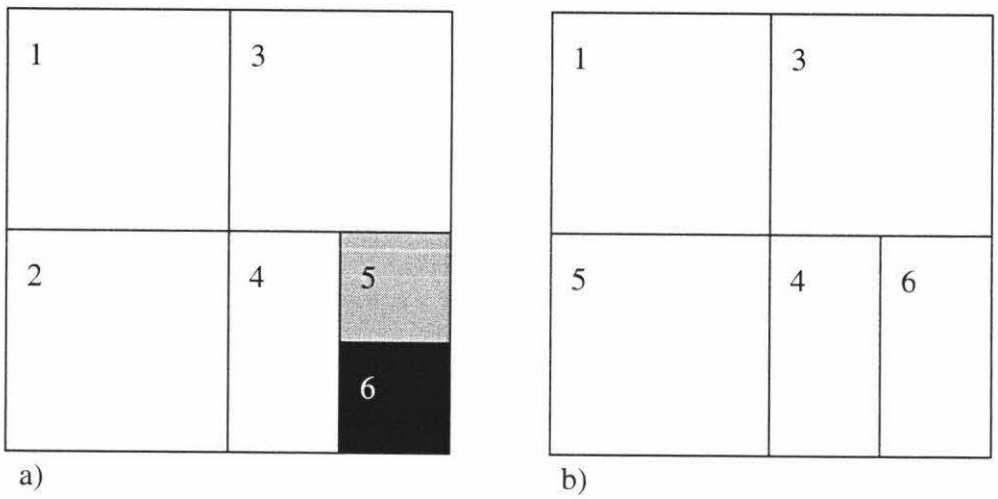

Figure 1-2 Merging of CAN zones during departure

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.

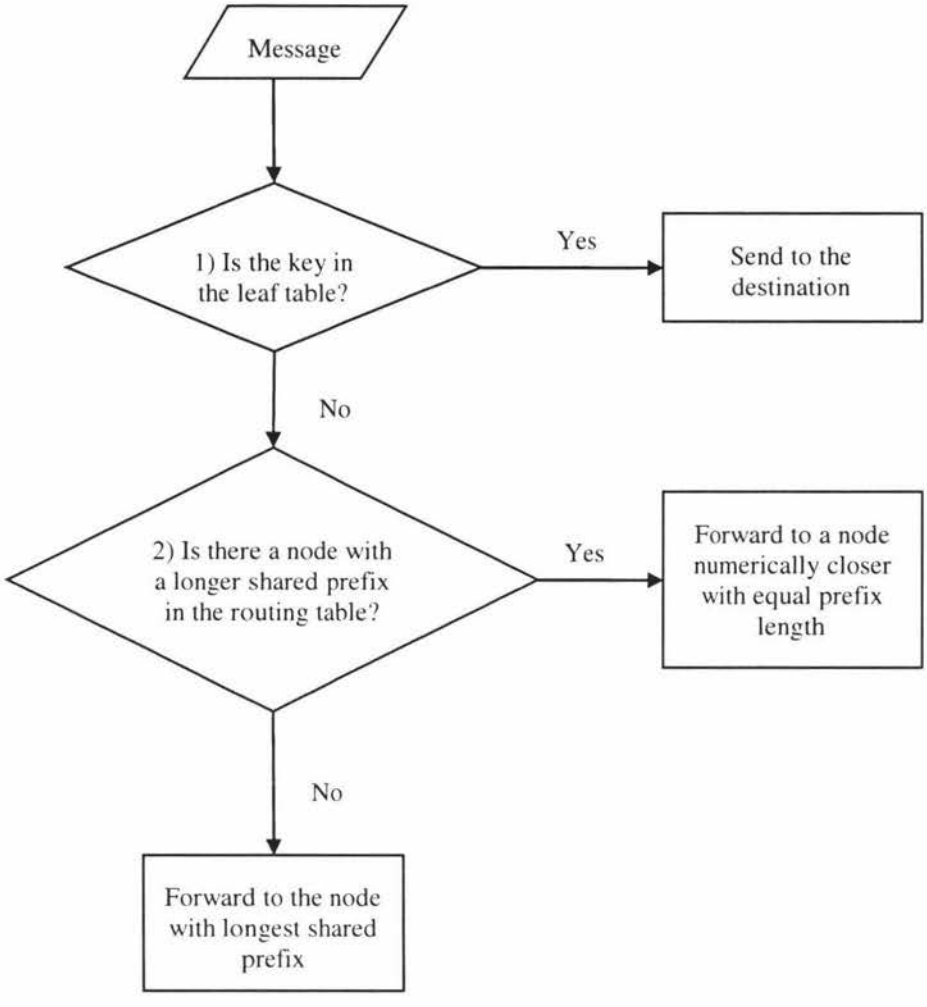

Figure 1-3 Pastry routing flowchart

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.

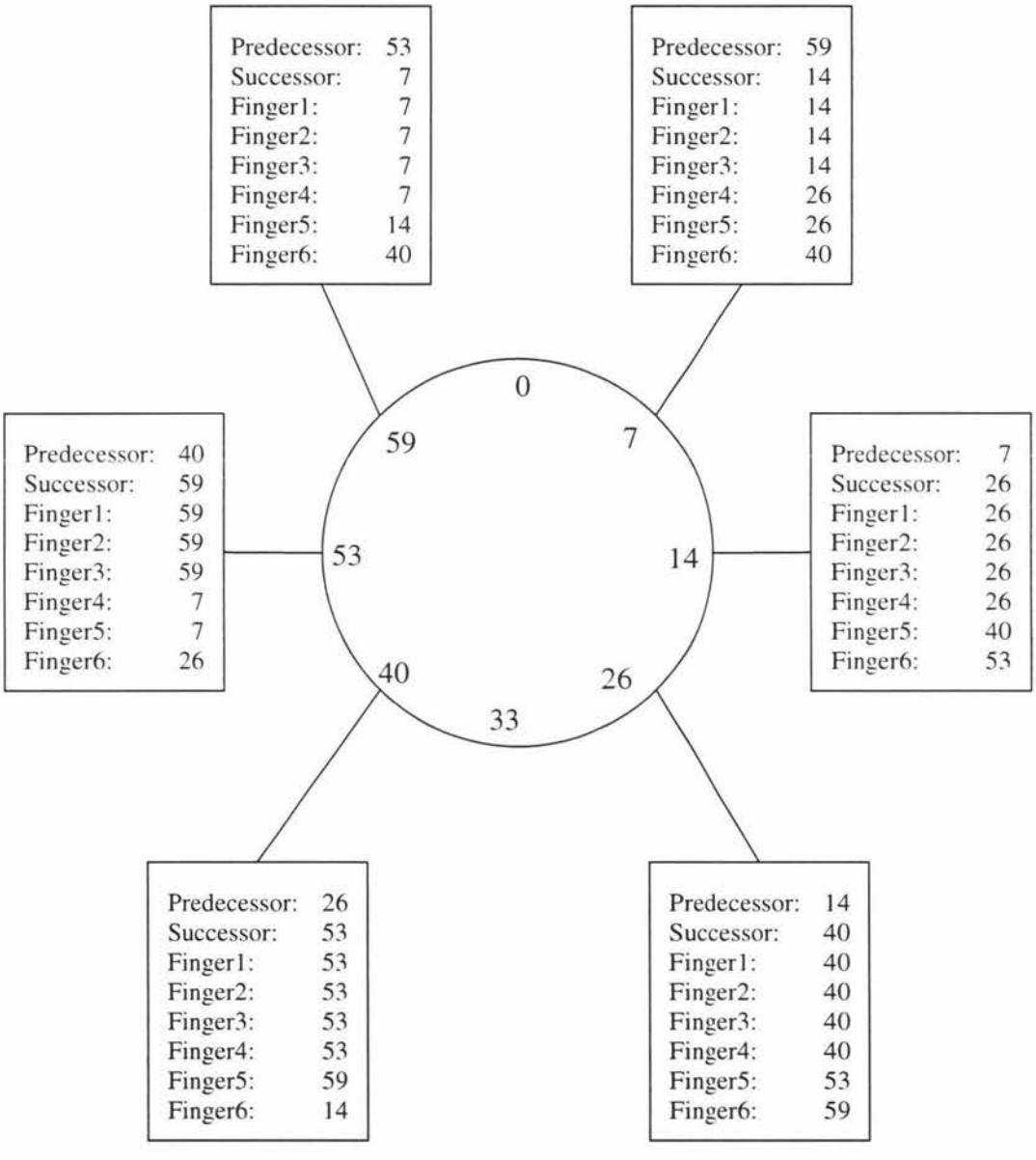

Figure 1-4 Chord routing

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.