The Dynamic Construction of Clusters from Idle Workstations Using Active Networks

by

Rajendra Singh

A thesis
presented to the University of Manitoba
in partial fulfilment of the
requirements for the degree of
Master of Science
in
Computer Science

Winnipeg, Manitoba, Canada, 2003

THE UNIVERSITY OF MANITOBA

FACULTY OF GRADUATE STUDIES

*****

COPYRIGHT PERMISSION PAGE

THE DYNAMIC CONSTRUCTION OF CLUSTERS FROM IDLE WORKSTATIONS USING ACTIVE NETWORKS

BY

RAJENDRA SINGH

A Thesis/Practicum submitted to the Faculty of Graduate Studies of The University

of Manitoba in partial fulfillment of the requirements of the degree

of

Master of Science

Abstract

This thesis presents an active-network-based system that supports the dynamic creation of clusters (collections of machines that act as a single computing resource to perform computations in parallel) to satisfy the computational service requests of users. Active network nodes (routers) are used to match service requests with stored measurements (of CPU type, load, etc.) to make a forecast of available resources (machines) to meet requests. Selected resources are then combined to form the clusters.

The active routers are organized in a hierarchical manner and perform the job of gathering the load information and storing it in their "soft-stores" in circular queues ordered by date thereby enabling easy prediction. Routers also forward their load information to their parent routers so each router stores information for all *the machines beneath it. When a service request arrives, a router makes a forecast of the resources available, selects sufficient resources to construct the requested cluster and returns a list of the selected resources to the requester. If insufficient resources exist, the request is sent to the parent router, which has access to more resources.*

Using this approach, the routers are in an ideal position to both gather measurements and make "scheduling" decisions and certain distinct advantages are achieved. These include anonymity of service, load distribution and improved fault tolerance.

Keywords Active Networks, High Performance Computing, Clusters, Computational Grids, Resource Discovery, Resource Allocation, Daemons.

Acknowledgements

I would like to thank my supervisor Dr. Peter Graham for all the ideas, motivation and encouragement throughout this research. He was a constant source of support, without which this thesis could not have been compiled. I really appreciate the way he helped me organize the research work, which not only ended in the completion of my thesis but also in learning the right direction of setting things up.

I am grateful to my parents (and all my relatives) for all their encouragement and for extending their support always, without which I would not be what I am today. I would also like to extend my gratitude to my sister and brother-in-law for having put the insight into me of obtaining higher education in Canada. I love my nieces Deeksha and Divya.

I would also like to thank Mr. Farook Mohammed (former student of Dr. Graham) for recommending me to Dr. Graham. I am also grateful to all my committee members for having provided a good environment during my defense, and for all the valuable suggestions on my thesis draft.

I would also like to thank my friends and roommates for extending their support at all times.

Above all, I would like to thank the God Almighty for this wonderful life.

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

1.1 Introduction

The use of High Performance Computing (HPC) [BM98] is important to advanced research and development in many industries and institutions. Critical problems that are computationally intensive can only be solved when high performance computing is used. HPC, generally, requires many processors to operate in parallel, the availability of large disk space and memory, and custom applications.The use of traditional parallel machines (e.g. the SGI Origin Series, the Sun Sunfire Series, etc.) is very costly and thus is not always feasible. A cheaper alternative is the use of cluster computing. A cluster consists of many conventional machines (e.g. workstations or PCs) connected via a high speed switched network that can cooperate to efficiently perform many computational jobs in parallel. Such clusters are normally built from collections of *dedicated* computers and therefore the use of a cluster requires a capital expenditure to purchase the computers and network infrastructure that connects them.

A huge pool of compute cycles and other significant resources attached to existing

machines are commonly under-utilized in many organizations. These could also be used to meet the demands of high performance applications if they could be brought together in the form of large-scale clusters. The research presented in this thesis addresses the construction of such clusters dynamically using "Active Networks".

Active networking allows certain computations to be performed in the network itself rather than on host nodes at the "edge" of the network. Active code is automatically installed and executed at routers whenever and wherever it is needed. In the research described in this thesis, active networking is used to collect information on computational resources (i.e. workstations/PCs) that are available in a networked environment and to service requests to form clusters from these resources. Network routers are in an ideal position to gather and use such information and this approach offers advantages over centralized, host-based techniques where a single processor might provide such services. First, such a centralized host represents a single point of failure that could bring the whole system to a halt. Further, a single processor (and its surrounding network) may also be easily overloaded and the probability of overloading grows with the size of the distributed system. Finally, since the active code runs inside the network there is no need for any machine to be aware of the identity of other machines. This isolation simplifies algorithm construction [Gra01].

1.2 Organization

The rest of this thesis is organized as follows. In Chapter 2, work related to that discussed in the thesis is presented. This includes material on computational clusters and grids, distributed resource discovery and management systems, active networks generally and ANTS (Active Network Toolkit System) specifically. Chapter 3 describes the problem and motivates an active networks solution. An active networks based mechanism for

the dynamic construction of clusters (using ANTS) is presented in Chapter 4 and its implementation details are discussed in Chapter 5. An assessment of the practicality and applicability of the technique is made in Chapter 6. Chapter 7 presents conclusions and directions for future research.

Chapter 2

Related Work

This chapter first describes parallel computing generally, dedicated compute clusters specifically, and the concept of computational grids for parallel computing on a grand scale. The Network Weather Service, NetSolve and the Condor system are then reviewed as examples of systems that discover and manage resources in distributed computing systems. A survey of different Active Networking concepts and environments is then presented followed by a more detailed discussion of ANTS (the Active Network Toolkit System), the active network environment used in this thesis.

2.1 Parallel Computing

Parallel processing is the concurrent execution of a program/job using multiple processors. The program is divided into pieces that are supplied to different processors for execution. The processors perform their part of the job and return their result. These results are then combined to obtain the final (cumulative) result. Parallel processing is important because it offers:

- ⋆ *Increased Speed*: Obviously, a job running on multiple processors with each processor executing its part of the job is potentially much faster than a job running on a single processor, and

- ⋆ *Support for Computationally Intensive Problems and Large Data Size*: Large computational problems exist in many organizations that can only be solved in a reasonable time using parallel processing and the large resources (e.g. memory) that come with it.

Granularity refers to the size of each part of a parallel job. Granularity plays a role in determining the effectiveness of parallel processing and so does inter-process communication. Generally, the grain-size of each program piece should be approximately equal. A significantly larger section of program alloted to a specific processor may degrade the performance of the program due to the delay incurred by the processor with the larger portion of the program. Due to the distribution of a single program across several processors, the communication speed between the program parts will also affect the efficiency of executing a job in parallel. If too much (relatively slow) communication must take place then the parallel processes will spend a lot of time waiting for communications to complete and performance will suffer. This often occurs when the grain size is too small.

Existing parallel programming tools such as MPI [Pac98] and PVM [Gei94] allow programmers to write message passing based parallel programs via a library of messaging routines built on top of TCP/IP or other network protocols.

MPI, the "Message Passing Interface" standard, is used widely in the parallel processing research community. The MPI library supports message passing constructs that allow sending/receiving of messages for communication in parallel environments between arbitrary machines involved in performing a job in parallel to enable high performance computing not only on massively parallel machines but also on workstation clusters. The

PVM (Parallel Virtual Machine) software also enables programmers to use collections of heterogeneous computers together to solve computational problems in parallel, this time by calling PVM library routines.

2.2 Dedicated Compute Clusters

"Beowulf" [Bec95, Rid97] style systems form dedicated compute clusters, commonly incorporating commercial PC's connected by commodity networks with a dedicated host machine that provides and controls access to the cluster. Beowulf systems are based on the Pile-of-PCs [Bec95, Rid97] approach which is similar to the earlier Cluster of Workstations (COW) and Network of Workstations (NOW) [Cas94, And95] approaches. In these systems commodity PCs are coupled together so they can be used as parallel computers. Beowulf (and other) networked parallel systems are explicitly designed to support parallel programming at low cost without sacrificing the performance needed to solve a wide class of computational problems. Collectively, Beowulf systems also offer large aggregate memory and disk capacity (in addition to aggregate computational capacity) that rival much more expensive parallel machines[1].

Early Beowulf systems used "channel bonding" of multiple slower network links to provide high speed communication since sufficiently fast networks were not available. Channel bonding logically combines multiple network links together and distributes the packets among them uniformly thereby increasing the effective bandwidth. The use of multiple networks across the nodes is transparent to the users [Res96].

The original "Bonded Dual Net" Beowulf system [Bec95, Rid97] connected 16 ma-

[1]Thus, for example, an application which has a very large set of data to be processed can easily buffer that data on the combined disk space available and then process it efficiently using the high aggregate I/O bandwidth of the multiple machines.

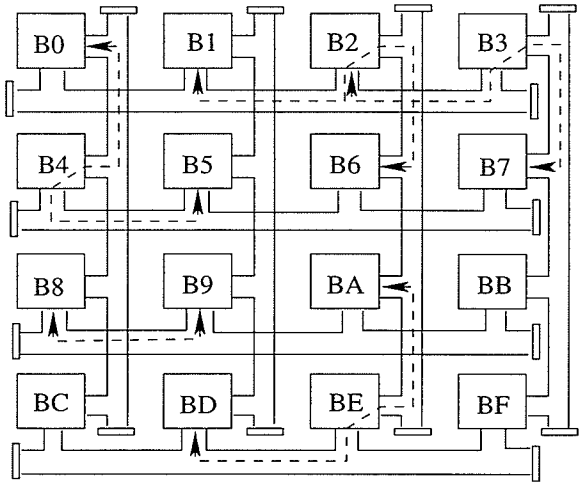

Figure 2.1: Switched Network Configuration (adopted from [Res96])

chines to form a cluster by using a pair of Ethernet buses giving 20 Mbps of shared network bandwidth. These Ethernet buses were used across the entire cluster to connect every machine and worked in parallel as a single virtual bus. Due to the bandwidth limitations of using a single bus, alternative network topologies were designed that connected the 16 machines using 8 Ethernet segments to form a $4X4$ two-dimensional mesh where every Ethernet segment connected 4 machines. Four Ethernet segments were used to connect the machines vertically and four were used to connect the machines horizontally. Using the original Bonded Dual Net topology every node has a direct route to all other nodes. In the alternative "Routed Mesh" topology (also known as the Software Routed Network Configuration), however, each node is capable of communicating with only six other nodes (three in the node's column and three in its row). The other nine nodes require communication using an intermediate node as a router. This topology is illustrated in Figure 2.1.

Another alternative to the original topology used a variation in the interconnection of the network segments as shown in Figure 2.2. This variation incorporated two Ether-

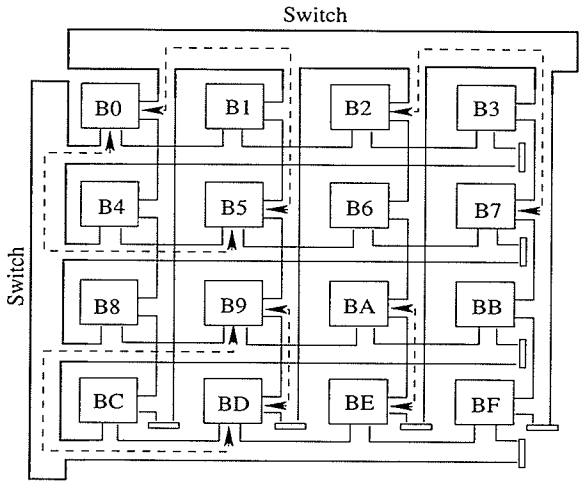

Figure 2.2: Switched Network Configuration (adopted from [Res96])

net switches with four-ports each, which connect the network segments with each other vertically as well as horizontally resulting in an increase in potential bandwidth of four. Modern Beowulf clusters use more advanced, switched networks including Fast Ethernet, Gigabit Ethernet and such special, low-latency networks as Myrinet [Bho98] and SCI [Wal92].

Dedicated compute "clusters" [Bec95, Raj99a, Raj99b] (including Beowulf Systems) provide a good solution for solving many computational problems but the cost of investment still makes it difficult for some potential users (e.g. small research groups) to build such systems. With the deployment of high speed switched local area networks it is now possible to collect unused resources from non-dedicated and often largely underused machines situated at different locations to form ad-hoc clusters that can be used to perform computational tasks. This form of clustering provides computational power at extremely low cost and still provides reasonably good performance. Exploiting these resources via dynamically constructed clusters is the focus of this thesis.

2.3 Computational Grids

The term "The Grid" came into existence in the middle 1990's to denote the construction of a Nation-Wide computing infrastructure in the form of a very large scale distributed computing system built by connecting geographically distributed machines and other network resources available at various institutions and organizations[2]. Foster, et al [Fos99] argued that the ever existing need for more compute cycles to solve ever harder compute intensive problems can only be satisfied with the power of a national-scale computational Grid.

Such Grids not only provide the huge aggregate power of their available computing resources but also the data and other significant resources that are contributed by the participating machines. Grid computing therefore must deal with the problem of the provision and co-ordination of *general* resource sharing across various multi-institutional "virtual organizations" that are trying to concurrently solve computationally intensive problems [Fos01].

Each such virtual organization consists of individuals, departments and other groups who are willing to contribute their resources and who are also in need of more resources themselves for solving their own compute intensive problems. By sharing resources, each participant may, at certain times, have access to much larger resources than they would otherwise have access to. Sharing rules in the Grid strictly define what can be shared, when, and with whom. The Grid must also provide services and tools for managing scalable virtual organizations.

The Grid Protocol Architecture [Fos99] is layered like the Internet Protocol (IP) Architecture. The grid protocol architecture consists of: "Fabric", "Connectivity", "Resource", "Collective" and "Application" layers. The Fabric layer provides raw resources

[2]The name is by analogy with national *power* grids.

such as computational capacity, storage, network links, etc. which users have agreed to share using the Grid sharing protocols. The Connectivity layer is the communication and authentication provider. To exchange data between resources provided by the Fabric layer, a communication protocol is required. To maintain security in a Grid system, authentication is required for the users as well as for the resources offered by the Fabric layer. The Connectivity layer provides both communication and the needed authentication protocols to support secure network transactions[3]. The Resource layer implements protocols for secure negotiations, monitoring, control, payment, etc., of the resources used. It is built on the Connectivity layer protocols and is meant to deal only with a single resource. Multiple resource co-ordination is handled by the Collective layer protocols. Finally, the Application layer consists of the applications themselves which actually run in the environment provided by the virtual organizations using the facilities provided by the Collective layer.

Different software systems have been developed to support Grid computing. Pre-eminent among these are Globus [Fos97] and Legion [Gri94]. The Globus toolkit uses the existing Internet protocols and a Public-Key based Grid security infrastructure that provides secure communication and authentication to support Grid computing. Globus offers a programming environment that provides general services that users can pick and choose from to meet the requirements of their applications. These general services are available in the form of a code "toolkit" that is used for developing applications.

Unlike Globus, Legion is totally based on the Object-Oriented paradigm and thus offers a tightly integrated Grid programming environment. This provides uniformity and transparency across the heterogenous platforms but at the cost of imposing a uniform programming paradigm on all applications.

[3] An example of such a protocol is the *Single Sign-on* [Fos01] protocol, which allows the user to log on with an authentication check and then use the resources offered by the Fabric layer.

A fundamental requirement in any Grid-based system is to be able to locate available resources that applications can then make use of. This process is known as "Resource Discovery" and must be followed by "Resource Allocation", which assigns discovered resources to specific applications. This aspect of grid computing is also applicable to dynamically constructed clusters as discussed in this thesis.

2.4 The Network Weather Service

The "Network Weather Service" (NWS) [Wol97, Wol99, Wol98] is a system that gathers information on network and host load. It uses this information to forecast the expected availability of network and computational resources in the near future. NWS is implemented as a centralized system that consists of processes running on several machines. There are four core components that constitute the NWS. The first is the *Persistent State*, which actually stores the load information that is collected. The *Name Server*, maintains information about all the processes using NWS and the communication protocols involved. *Sensors*, gather measurements over the network while running on participating host nodes. The *Forecaster*, predicts the likely future availability of resources in response to user requests for such information.

The NWS uses different processes (Sensors) for gathering the network resource measurements and the CPU load measurements. The information gathered is properly tagged and stored in the persistent state, where the forecaster can retrieve the information to make forecasts of future availability and expected performance of the resources.

A problem with the NWS is that if a critical process such as a service provider (e.g. the *Persistent State* or *Forecaster*) fails, then the whole system comes to a halt (i.e. a single point of failure causes a breakdown). Furthermore, since a single process controls the information for all the machines, it is likely that when the system scales over a

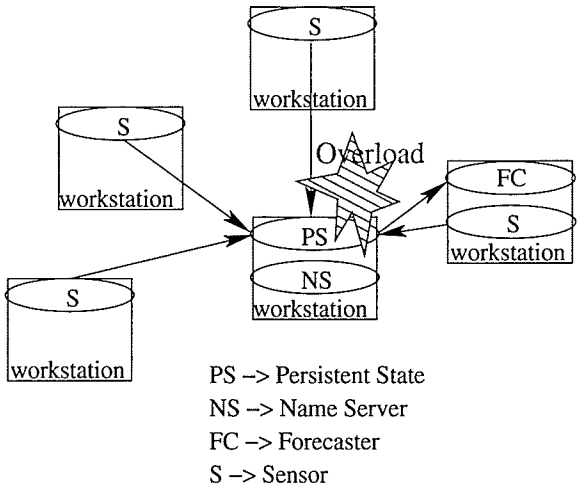

Figure 2.3: The Network Weather Service [Wol99]

large network of workstations, that process will become over-loaded (as shown in Figure 2.3). Ultimately, this may also result in network congestion and lead to a significant slowdown of the system. Additionally, the NWS also needs to know the host ID's of the workstations participating in the system and they, in turn, need to know the host ID's of the *Persistent State* and *Forecaster* servers. This limits the system in terms of scalability and makes failure recovery difficult since host ID's may change. The research presented in this thesis address all of these problems.

2.5 NetSolve: A Network Server for Solving Computational Science Problems

"NetSolve" [Cas96] also attempts to use widely distributed computational resources for solving computational problems but at a higher level. Unlike many other systems, it is oriented towards the use of existing applications remotely not the creation of new distributed ones. NetSolve provides an efficient way of handling load-balancing and

strives to offer the best currently available resources on the network for solving each application problem.

NetSolve is client-server based, with several different interfaces that have been developed for the convenience of the users of different problem solving environments. This approach is promoted as being better than just providing programming libraries for different platforms, as it saves non-technical users from having to devote time to learning how to use the libraries. NetSolve also supports heterogeneous environments and can be used locally or over wide area networks.

A NetSolve *client* contacts a NetSolve *agent* to locate resources. The NetSolve agents choose the "best available" computational resources based on the criteria of smallest expected execution time for a given problem[4]. NetSolve uses the standard TCP/IP protocols and XDR [Sun87] to permit communication between heterogeneous hosts over IP networks.

For ease of use by its users, NetSolve provides three interfaces: an *Interactive Interface* (completely free of code writing), a *MATLAB Interface* and a *Shell Interface*. Other programming interfaces have also been developed for more sophisticated programming language-based users.

2.6 Condor

The Condor scheduling system [Lit88, Bri92] was originally designed to work in a network of workstations environment. It schedules jobs onto workstations depending on the activities of the people who own the workstations. Idle workstations are detected

[4]This is computed using a custom performance model where every agent has its own view of the system [Cas96].

and jobs are scheduled onto them. The Condor system removes jobs and re-schedules them to other workstations when the owner resumes using their workstation (with the guarantee that the job will eventually be completed). The purpose of the Condor system is to maximize the use of a workstation environment's computing resources but it is not focused at all on supporting parallel programming (as in cluster computing).

The Condor system provides three fundamental functions to make efficient use of the computational capacity in a network of workstations: the analysis of workstation usage patterns, the design of remote capacity allocation algorithms, and the development of remote execution facilities. Condor uses the Up-Down scheduling algorithm [Mut87] for capacity allocation and the Remote Unix (RU) [Lit87] facility is used for executing jobs remotely. A critical feature of RU is check-pointing. RU creates checkpoints for recovery in case, at any point in time, the job fails to execute or has to be removed from the workstation. Each checkpoint saves the most recent state of the job so that once relocated, the job can be re-started from the saved state rather than from scratch.

Condor's ultimate goal is to locate workstations which are free and to allocate long running jobs that require very little user interaction to them. A Condor user need not know where each job is being executed as the system provides complete location transparency.

Condor works with a centralized, static coordinator but is otherwise highly distributed. It is the central coordinator that locates the idle workstations and allocates jobs to them. Every user job is registered with the central coordinator. Workstations taking part in the system have a local scheduler and their own job queue. It is the responsibility of the local scheduler to schedule the jobs on that workstation, but which workstation is to receive a Condor job and how much capacity is to be allocated is decided by the central coordinator. The central coordinator itself runs on some workstation and uses the Up-Down algorithm to priorize the workstations participating in the system. It manages the

available remote processing capacity in such a way that it can be fairly allocated among users, without being biased to users trying to use all the free cycles. Every workstation participating in a Condor system has an initial priority, which is later changed by using a Negotiator that negotiates the priorities among the workstations willing to execute the jobs, on behalf of the central coordinator. Different priorities may be assigned to different groups of users thereby allowing workstation owners to favor one group of remote users over another and/or to block access to specific/other users entirely.

The Condor system is focused exclusively on running collections of serial jobs and is therefore, not itself, appropriate for cluster environments. Some of its underlying concepts will, however, be applicable to the research presented in this thesis.

2.6.1 Flock of Condors

The "Flock of Condors" system [Epe96] is an approach to optimizing the use of resources in collections of workstations beyond the boundary of a single Condor system's "pool". Several Condor pools are connected via a wide-area network to create a *flock* of Condor systems. A *Gateway Machine* in each Condor pool manages the idle workstations in the pool and also handles the transfer of jobs across pool boundaries.

The flock approach helps to solve the *wait-while-idle* (WWI) problem by extending the boundaries of a conventional Condor system. The WWI problem refers to the situation where long batch jobs have to wait for machines in a local pool to become free while there are other resources available "nearby" that could service the job's needs. A Condor flock extends beyond a Condor pool's boundaries so that jobs submitted to one pool (the *submission* pool) can have access to the resources in another pool (the *execution* pool). This is made possible by the *Gateway Machines* (GWs) which manage the connection points between two pools. Each GW advertises the resources of its pool to other pools

and uses advertisements from other pools to decide when to transfer jobs to those pools (all subject to security issues and owner's rights.).

2.7 Active Networks

"Active networks" [Cal98, Ten96, Wet98, Wet99a, Wet97, Ten97, Wet96] are an approach to making the network more "intelligent". They facilitate the construction of new, network-centric applications by allowing users to inject customized programs into the network. These customized programs exist as "active code" in the active nodes (i.e. routers) of the network and the code is executed whenever it is needed. Active networks can also use the Active Network Encapsulation Protocol (ANEP) [Ale97] protocol for communication in an IP environment by encapsulating active "capsules" in IP packets.

A number of active network systems have been proposed and constructed. These are briefly reviewed before considering the ANTS [Wet98] system in greater depth.

2.7.1 ANTS: The Active Network Toolkit System

"ANTS" was developed specifically to be able to easily and quickly deploy new services into the network. The idea behind developing such an architecture was "to support the use of new protocols based on the agreement of the two (or more) parties participating rather than having a centralized agency, which registers all the protocols and forces all participating parties to use only registered protocols" [Wet97].

The "ANTS" architecture is an extensible Internet architecture. It was designed in such a way that the packets, which are termed "Capsules", actually carry the active code. Active code is simply a program defined by the user that is used to create a customized

network protocol. Active packets are processed and forwarded depending on the execution of the "forwarding routine" the capsule is carrying. Each capsule may also carry a reference to additional code which should be executed when the capsule reaches an active node. If the node doesn't have that code, it can request the code from other nodes dynamically using a "demand-pull" approach (described later).

An ANTS based Active Network consists of the active nodes (application aware routers), channels through which the capsules can be sent and received, and the active applications. All the active nodes are connected to other nodes using a link layer channel. Being able to define new capsule formats and having the ability to specify the code that determines each capsule's behavior at the active nodes provides users the ability to customize the in-network processing that is done. Thus, capsule programming provides not only the ability to customize packet forwarding behavior, but also to provide more general computations (e.g. dynamically changing the payload contents while a capsule is traveling in the network).

Not all the nodes in a given network are necessarily active and capable of recognizing active capsules. Nodes that are not ANTS-specific, however, can accept active capsules embedded in non-active packets and will simply forward them towards their destination without active processing. An ANTS capsule header is recognized by the active nodes using a higher level protocol such as IPv4 or IPv6 protocol identifiers. Only active ANTS nodes, however, look within the IP packet to process it as an active capsule. This makes ANTS networks compatible with existing IP networks without introducing a new layer [Wet99b].

How does an active node know which code or forwarding routine should be executed when a capsule arrives? For this purpose the ANTS capsule format contains a type field, which specifies the forwarding routine to be executed. Furthermore, the value of this type field is actually an encrypted fingerprint (using MD5 [Riv92]) of the associated code to

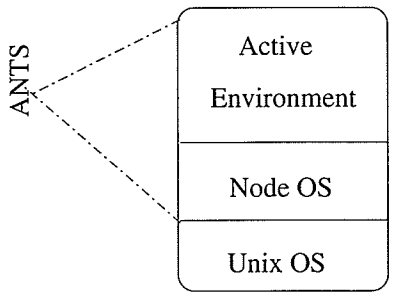

Figure 2.4: ANTS Environment

be executed thereby providing a degree of security against forged and/or modified packets. The capsules, when injected into the network from an end-node, are associated with a forwarding routine, a code group, and a protocol via the type field. A code group consists of a group of related capsule types that are distributed to active nodes as a single unit while a protocol consists of one or more code groups. All data defined in a protocol is shared by only the forwarding routines of the code groups belonging to the protocol. Thus, forwarding routines belonging to different protocols cannot interfere with the processing of other capsules at any active node. The type field is used the same way as the IP version and protocol fields are used in conventional networks: for demultiplexing the capsules to the appropriate forwarding routines they will use.

ANTS provides an "execution environment", rather than a node operating system. ANTS is specifically designed to run on top of the Unix operating system (see Figure 2.4) which supports the forwarding of capsules (in packets) efficiently. The Node OS always keeps track of the processes and execution of forwarding routines. It doesn't allow a forwarding routine to run for an excessively long time and also keeps track of which forwarding routines are available at the node.

When an active node receives a capsule it de-multiplexes the capsule to identify the appropriate forwarding routine and then checks for the availability of the corresponding

forwarding routine. If the forwarding routine is not available at the node, the active node sends a request to the previous active node visited by the capsule (it gets the address of the previous active node from the "previous address" field in the capsule). The chances of the corresponding forwarding routine being available at the previous active node is high because the capsule was forwarded only after the previous active node executed the required forwarding routine. In response, the previous active node sends the corresponding forwarding routine to the requesting active node. This mechanism of getting the code is an exclusive feature of ANTS and is known as "demand-pull" code distribution.

ANTS is the active networks system that is used in the prototype presented in this thesis. ANTS was selected due to the simplicity of deploying code in the network it provides, the fact that it is Java based and because ANTS supports the easy configuration of hierarchical network topologies (which are assumed later in the thesis). Although heterogeneity is not considered in the prototype, ANTS supports heterogeneous networks and this will make it possible to integrate heterogeneity into the prototype in the future. Furthermore, the ANTS approach is compatible with conventional network protocols which will make it possible to deploy the system in a real network at a later date.

Additional details about ANTS will be provided in Section 2.8 after other active network systems have been reviewed.

2.7.2 PLAN: Packet Language for Active Networks

"PLAN" [Hic98] stands for "Packet Language for Active Networks". PLAN describes a language which provides a programming environment for active networks. It is based on a typed Lambda Calculus which provides a restricted set of primitives and data-types. "Chunks" (*Code hunks*) provide the basis for executing PLAN code at active nodes.

PLAN has a two level approach to active node functionality. The first level is the

PLAN Level, which consists of PLAN language programs. The execution of these programs facilitates the discovery of resources without any authentication requirements. Once the resources are discovered, the *Service Level* provides authentication. The Service level manages the protocols and available network services to maintain a secure active network environment. PLAN (the language) effectively works as network-level "glue" in the sense that every PLAN packet contains a PLAN program which calls existing internal node service routines thereby constructing an active program "glued together" from the available routines. The presence of the customized "program" in the packet itself, which can be evaluated at each active node, is what makes the network active.

PLAN provides both security and certain guarantees of correctness. The formal and strongly-typed nature of the language makes PLAN programs pointer-safe [Hic98], guarantees that concurrently running PLAN programs are executed mutually exclusively and that the programs will terminate. Additionally, PLAN provides specific error handling, which is controlled by the *handler* field in each PLAN packet. Each handler field specifies a service name to be invoked if a certain exception is not handled properly when the program is executed. PLAN also provides good performance. PLAN's designers realized that performance would decrease if all PLAN packets had to be authenticated so PLAN was designed to be light-weight. Heavy-weight features, if needed, can be accessed via node level services rather than being included in PLAN packets.

Each PLAN packet is injected into the network with its *evalDest* field (described later) set to indicate the "evaluation" destination. Each packet also specifies a routing service function (named in the *routFun* field) and a resource bound specified in the *rb* field. The service routine named in *routFun* takes *evalDest* as a parameter and computes the next hop in the route. The *rb* field is decremented at every hop to make sure that the packet is discarded if its resource bound is exhausted (i.e when $rb == \emptyset$).

PLAN code can be developed using two specific languages: OCaml [Ler00] a language that provides functional, imperative and object-oriented programming facilities all together or Pizza [Ode97] which is an extension of Java.

2.7.3 PLANet: An Active Internetwork

"PLANet" [HMA+99] is an internetwork system which, unlike ANTS, does not rely on the existing IP [Pos81] infrastructure. Instead, it implements its own network layer services over whatever link layer is provided by the existing network infrastructure. PLANet is, of course, active because all its packets are constructed using PLAN. All the packets in PLANet contain programs and the functionality performed by each router can be dynamically changed, if needed, by dynamically loading additional code (active extensions). The routing behavior of each PLAN packet depends on the PLAN program used, which decides if the packet is to be evaluated at every node or simply be forwarded towards its destination (chiefly for performance reasons). The dynamically loaded active extensions, are written using the OCaml language and provide the nodes with common services such as address resolution, routing, etc. The PLAN programs determine the behavior at nodes. When a packet reaches its evaluation destination, the PLAN interpreter evaluates it and calls for the service routines required.

The PLANet node architecture consists of the existing link layer interface, an active extension containing a PLAN interpreter and routing extension, and the PLAN program itself in the active packet. Active packets destined for a particular node, when received from the link layer interface, are evaluated by interpreting the PLAN program and also, if needed, by calling the node's active extensions.

PLANet, naturally, also uses the PLAN packet format, with the *evalDest* field indicating the destination for evaluation of the code contained in the packet. A *source* field

indicates the node from which the packet originated. PLANet uses a 48 bit address assignment scheme for the node address and uses 16 bit port numbers. A modified form of ARP is used for address resolution. As PLANet uses the PLAN packet format, it has a *chunk* available in the packet, which contains the program and its initial parameters. The additional packet fields are *session*, which is similar to the protocol field of the same name in a standard IP packet, and *flowId*, which is the packet flow identifier.

2.7.4 PAN: A High-Performance Active Network Node

"PAN" [Nyg99] is an active network system, which is described by its authors as being the first step towards the "practical" use of capsule based active networks. It provides high-performance by caching dynamically compiled capsule code, which provides performance comparable to conventional networks. Its capsule based nature, of course, facilitates carrying mobile code and running it on routers – something that conventional networks do not support.

PAN offers a number of unique features. First capsules are processed in the kernel and not in user space. This significantly reduces the copying of data between kernel space and user space. Using clever memory management, PAN eliminates the need for nodes to copy the code of those capsules that are meant to simply be forwarded to other nodes and not to be executed locally. The node environment provided on each PAN node also caches the compiled capsule code so that it does not have to load the same capsule code and compile it again. This significantly reduces overhead and increases the overall performance of each PAN node.

In other ways, the architecture of PAN is quite similar to ANTS and PLANet. Network links connect nodes and communication between the nodes is accomplished by sending capsules. A user application uses the capsules to carry both the data/payload

and a reference to the code that needs to be executed at one specific (or all) routers for processing or forwarding, respectively. Demand loading is also used when code is not available at a specific router.

The basic components of a PAN node include three things. First, the "Software Segment" interface, which is responsible for memory management to minimize data copying and to provide shared access to the compiled code. Second, the "PAN Node Interface" which provides an interface to node services such as data structure access, local time query, etc. Third, the "PAN Application Library" which facilitates the receiving and sending of capsule and finally, a "Code Object" which actually contains the executable code and data. PAN assigns each code object a name using a cryptologic hash scheme, which ensures that the code executed in-network is exactly what is meant to be executed for a particular capsule. Each router node also maintains a soft store for application code and data.

The performance of a PAN node is reported to be comparable to that of a conventional network node. Tests show that by using native object code, a PAN node can forward 100 Mbps of data using a 200 MHz Intel PentiumPro running Linux. The experimental results also show that a PAN active node executing native object code in the kernel space can actually overload a 100Mbps fast Ethernet network with 1500 byte packets and the processing time of each packet is only 13 percent more than for a conventional router [Nyg99].

2.7.5 Netscript

[Yem96] describe an approach to programmability of network nodes using NetScript (a multi-threaded language designed for communications and building networked software on a programmable network) and Delegated Agents [Yem91]. The *agents* are the

programs that can be dynamically deployed and executed at different intermediate nodes. The NetScript project considers the network as a NetScript Virtual Network (NVN) comprised of several Virtual Network Engines (VNE's) connected by Virtual Links (VL's) analogous to the conventional network nodes and physical links. The programmability is deployed in a NetScript Network by programming the VNE's using NetScript Agents that process the packets received. Each VNE consists of an Agent Service Layer which is responsible for the execution of the agent programs as well as for controlling the communication between agents. The Connectivity layer manages the VL's between the VNE's. The multiple threads of a NetScript program executing an agent's program at various VNE's manage the relaying of packets belonging to a particular protocol. A NetScript packet contains a NetScript encapsulation header which is examined upon receipt at a VNE to determine the protocol to which the packet belongs and is de-multiplexed to select the respective program for processing.

NetScript was designed to provide enhanced network management capabilities and can be used to program existing network management applications (e.g. Remote Network Monitors and SNMP [Sta96]).

2.7.6 Smart Packets

"Smart Packets" [Sch99] were developed to improve Network Management by making the managed nodes programmable. There are several advantages offered by Smart Packets. First, the overhead on the management centers (the nodes that manage other nodes using SNMP agents, for example) can be reduced as they can send programs to the managed nodes for execution. Second, the managed nodes can run the programs without any communication with the management centers, which further reduces the load on the management centers. It is assumed that programs will abide by the rules and regulations that

are to be followed for managing the network and that such programs are self-sufficient for execution.

A virtual machine in every active network node initiates the execution of the code contained in each Smart Packet. Smart Packets use a special compressed representation of the code (that is to be executed at the managed nodes) for security purposes.

Smart Packets are encapsulated using the ANEP [Ale97] protocol for transmission in IP networks. A packet can be sent in the network in either *end-to-end* mode or *hop-by-hop* mode. The choice affects the evaluation of the contained code. End-to-end node transmission executes the program only at the destination while hop-by-hop mode transmission executes the program at each router along the way (i.e. after every hop). An ANEP Daemon process is responsible for injecting and receiving Smart Packets. Smart Packets use IP Router Alert Options [Kat97] to inform active network routers that a Smart Packet has been received and that they should execute the program contained in it. Conventional (non-active) routers ignore the Alert Option and simply forward the packet on towards its destination.

2.7.7 SANE: A Secure Active Network Environment Architecture

"SANE" [Ale98] stands for Secure Active Network Environment. SANE attempts to provide greater network flexibility (as all active network systems do) with a particular goal of providing more efficient routing. The argument given is that conventional networks (using IP infrastructure in which a router or switch receives a packet, examines the header, and simply forwards or routes the packet to its destination) may not provide ideal routing. If some programming liberty is given to the user then user code can route their packets as desired. Conventional packets depend on the routers to make a decision on where they should be forwarded to, even if the path they are to be sent along is con-

gested. If packets can be programmed so that they can select their path depending on resource information available in the network, then they can be routed more effectively than using conventional networks. The end result being improved network efficiency.

The SANE architecture is a layered architecture, where every layer depends on and trusts the layer below it. Active networks provide programmability of packets and give users enough liberty to customize the behavior of their packets in the network. This, however, increases the security risks. To maintain integrity and trust within the system, SANE uses both *static* and *dynamic* integrity checks when dealing with active code. Static integrity checks are infrequent and are only performed, for example, when the network is booting. The trust developed at this point, however, can only persist while the network state remains unchanged. Obviously, in a scenario where different kinds of resources and communication protocols exist the state has to change. At this point dynamic integrity checks are used to check the changed state (on a per-user, per-packet basis).

SANE also uses the AEGIS [Arb97] secure bootstrap architecture for booting. AEGIS provides verification of executable code using digital signatures. Modifying the BIOS on network nodes by supplying the verification code and public key certificate(s) accomplishes the job of secure booting. Once this lowest level layer completes the process of booting, the upper layers establish a trust relationship with it (based on the *static* integrity checks performed).

2.8 ANTS: A More Detailed Overview

Active Networking is all about making the network components more intelligent rather than just carrying data bits passively. An understanding of the abstractions and services

provided by the ANTS active network system will be necessary to understand the prototype implementation described in this thesis. Accordingly, this information is now presented in greater detail.

2.8.1 ANTS: the Active Networks Toolkit System

Recall that ANTS is a capsule based Active Network environment in which the capsules carry customized code (or at least a reference to it) from node to node. Compatibility with TCP/IP is ensured by designing the capsule format in such a way that conventional network routers can recognize each capsule as a packet and forward it to the next router using whatever default (non-active) is available. This is desirable since not every node in the network will be an active node. The ANTS capsule format ensures that conventional routing is never affected.

ANTS is designed to run on the Unix operating system, where configuration files can be customized to make the machine act as an active node (router) or as an end node (where user specific applications can be created for sending customized capsules into the network). The active nodes are connected to each other via channels which use the existing link layer channels, as shown in Figure 2.5. Capsules travel through the channels and are received by the active nodes to be processed. New protocols can be dynamically deployed simply by registering them with the active nodes. Such protocols are an agreement between only the communicating parties themselves and therefore do not require centralized control. New types of capsules must be constructed to carry the customized code that implements these protocols.

Active nodes run continuously and receive and re-transmit capsules after performing potentially significant computation on the capsules' data. Active nodes cache the frequently used code for processing capsules and if the code is not available they can re-

quest it from any node which the capsule has already visited. This is done using a "code transport mechanism" known as "demand-pull".

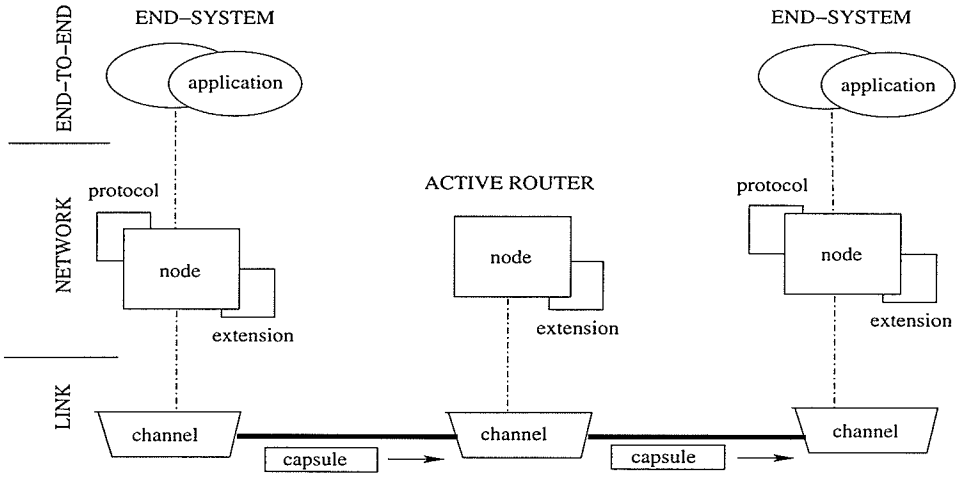

Figure 2.5: ANTS Network Structure (adopted from [Wet99b])

2.8.2 ANTS Architectural Components and Issues

Figure 2.5 shows the key components used in an ANTS system. This section describes those services offered by the ANTS components that were used in the thesis prototype.

Active Nodes

Active nodes in an ANTS active network receive capsules from link layer channels and process them. Such active nodes in ANTS can be either core nodes or edge nodes (but not both), depending on the purpose of the node in question. Each core node acts as an active router and runs any active code it receives. Each edge node is associated with a user application. The `Node` class of ANTS exports the node API to capsules, applications and extensions at runtime. The API exported to capsules enables each capsule

to be recognized at active nodes and facilitates the execution of their forwarding routines. An active node provides capsules with a set of service calls by which the capsule can access information about the active node at runtime such as the local time, local address, etc. Another function that is provided to the capsule is the ability to use the active node's soft state. By accessing the soft state of the active node, a capsule can store application-specific and processing-specific data for some period of time. Data objects in the soft store are guaranteed to be stored at least until the forwarding of the capsule is complete but normally persist longer, space permitting. The primary original purpose of storing such objects was to improve the performance of forwarding by caching useful data. Frequently used objects (e.g. those needed for capsule forwarding) can be stored and be re-used by subsequent capsules in their processing. The other services provided by active nodes directly support the routing of capsules.

When creating an Active Node an instance of the class `Node` is instantiated, which provides a secure Node environment for executing the forwarding routines of protocols. In a scenario where resources and protocols cannot be trusted, a secure environment is necessary to check the authority of routines and protocols to prevent, among other things, the unwanted consumption of network resources. The `Node` class provides an environment which maintains a registry of all valid protocols by providing service calls including *register()* and *unregister()* to applications which create and destroy active protocols. The `Node` class also manages the distribution of network resources among these protocols efficiently.

These API's facilitate applications and capsules having access to the active node's resources for the execution of their processing routines. At the same time, active nodes protect themselves by not providing full access to their resources and by maintaining complete information on, and control over, the processing routines and protocols that are executing on them and sharing their resources. For example, the primitives that

provide access to the active node storage by capsules, facilitate the storage of data for subsequent use by other capsules of the same type as the data's creator. This has the effect of increasing the forwarding rate of the capsules since significant computation for loading the required data for processing capsules is avoided. At the same time, if the processing of a capsule's forwarding routine exceeds a prescribed resource consumption limit, processing is terminated and the related data stored in the active node's soft store is deleted.

Other key primitives exported by the `Node` class include *routeForNode()* and *deliverToApp()*. Whenever a capsule arrives at an active node, a direct query can be made of the active node's address. If the active node's address is equal to the destination address then the capsule is delivered to the application using *deliverToApp()*. Otherwise the capsule is forwarded to the next neighboring node. This forwarding is performed using *routeForNode*, which fetches the address of the neighboring node from the routing configuration file (this file contains the information about the routes between network nodes) and sets it as the capsule's destination.

Capsules

One of the most important components in Active Networking, is the "capsule" which contains the active code ("A capsule is a generalized replacement for a packet" [Wet98]). ANTS uses capsules which are similar to packets in conventional networks. Capsules carry the payload data as well as code for, or a reference to, the forwarding routine that is meant to be executed upon reaching an active node. There can be several capsule types reaching any active node and making a distinction between different kinds of capsules is necessary for the proper function of the node. This distinction is achieved by defining *code groups* and *protocols*. Capsules of related types belong to the same code group and, further, similar code groups are collected together to form a protocol. This provides a

level of protection to the node execution environment by restricting forwarding routines belonging to different protocols from interfering in one another's processing.

src address	dst address	resource limit	version	type	previous address	type dependent (measurements)	payload

Figure 2.6: Capsule Format (adopted from [Wet99a])

The format of a capsule is shown in Figure 2.6. Each capsule contains the following fields:

- ⋆ *source* and *destination* addresses are equivalent to the conventional IP packet source and destination addresses.
- ⋆ *resource limit* is similar to the Time To Live (TTL) field of an IP packet and is used for resource management.
- ⋆ *version* is the ANTS version number.
- ⋆ *type* is the capsule type.
- ⋆ *previous address* is the address of the previous active node visited by the capsule.
- ⋆ Other header fields are type dependent.
- ⋆ *payload* contains the actual data being transmitted in the capsule.

The forwarding of capsules is dependent on the type of capsule. When a capsule arrives at an active node, the capsule header is examined. The type field is used to demultiplex the capsule to the appropriate forwarding routine. Forwarding routines are executed to handle the forwarding of each capsule after checking for the availability of the corresponding forwarding routine at the active node that received the capsule. If the

forwarding routine is available, it is executed otherwise it is requested from the previous node the capsule visited (using the previous address field). While the forwarding routine is executed, the node monitors the capsule's processing for security and performance purposes. A forwarding routine is allowed to execute only for a particular period of time. If the processing time limit is exceeded, the capsule processing is stopped and the capsule is then simply forwarded using the default route. When this occurs, the associated node also makes a note for further processing so that capsules of that type are subsequently forwarded using only default routing.

Each new capsule type is constructed by making a subclass of the `Capsule` class which is provided by the ANTS Toolkit. Certain "accessor" functions such as *setSrc()*, *setDst()*, etc. are used for manipulating the capsule's header fields. The methods *getCapsuleID()*, *getGroupID()* and *getProtocolId()* are used to determine the capsule's type, code group and protocol. The most important method for a capsule is *evaluate()*, which is used at every active node to execute the active code for forwarding purposes.

The ANTS Code Distribution Mechanism

Code Distribution is an important aspect in Active Networking that is not required in conventional TCP/IP Networks. Conventional networks, being passive, don't have a need to execute any kind of code except built-in services used to forward IP packets towards their destinations. In active networks, since active code may be installed at the nodes, there is a requirement for active nodes to be able to execute some kind of dynamically distributed code. As described in Section 2.8.2, capsules carry data as well as code or references to the code that needs to be executed at the active nodes. A node should have the code that is to be executed or, at least a mechanism to get the code when it is needed. ANTS uses a technique for transferring such code known as the *demand pull* mechanism, which is illustrated in Figure 2.8.2 and described below.

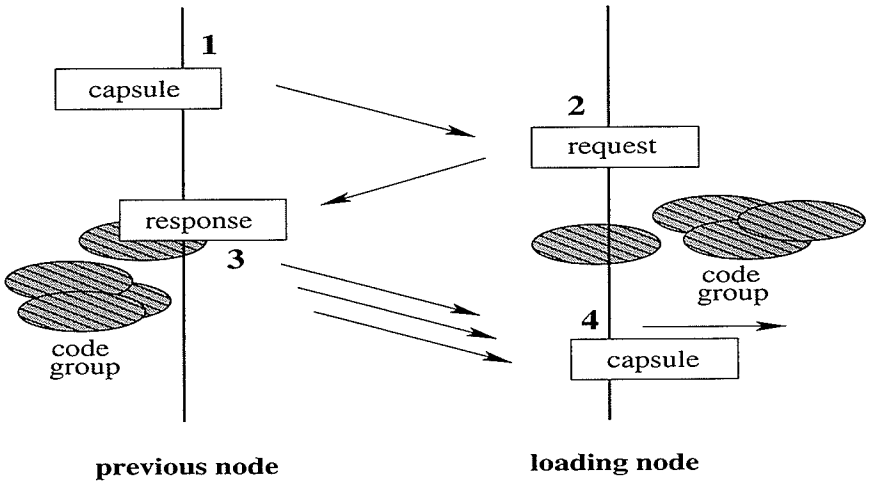

Figure 2.7: Demand Loading of Code Groups (adopted from [Wet99b])

Two extremes for code distribution are possible, as described by Wetherall [Wet99b], depending on the frequency of the use of active code. If the deployment of new services is rare then there will be infrequent need for new code and the code can be added to routers through normal software upgrades. If the new capsule types will frequently need to be processed, however, then it would be more appropriate to have the code carried in the capsules themselves. ANTS' demand pull mechanism lies between the two extremes since the frequency of capsules calling new services in ANTS is assumed to be relatively, but not extremely, high.

The demand pull mechanism actually starts at the end-systems, where an instance of the `Node` class exports classes to provide services to the network. The code that implements each new service is thus acquired by the application first. When an application acquires new code it has full code information including the forwarding routine, code group and the protocol that implements the service. An application registers the code with the local active node, which then performs a check for the availability of the corrspnding capsule type and is then ready to deploy the service into the network. At this

point, applications can send capsules that will use the service code into the network.

When a new, unrecognized capsule type arrives at a node within the network then the demand pull mechanism is used. For this purpose the node fetches the previous address field from the capsule header and generates a load request capsule and sends it to the previous node visited by the capsule. The availability of the code at the previous node is high since the capsule came from that node only after the required forwarding routine was processed at that node. The previous node, upon receiving the load request, generates a series of response capsules that contain the code group needed and sends them to the requesting node (shown in Figure 2.8.2). Upon receiving the response capsule(s), the requesting node re-assembles the code and then can use the associated code group to perform further processing. This process of transferring code is unreliable but very lightweight. There is a small but non-zero chance that the requesting node will never receive the response capsule(s). In that case the capsule will either be dropped or just forwarded using default routing.

Protocols

When developing a new custom capsule type the user is required to write a protocol and register it with the active nodes that will use it. A protocol, in general, is a mutual agreement between two or more parties concerning the communication patterns that will be used to permit them to cooperate. Thus a protocol facilitates each node recognizing and processing its related capsule types. ANTS provides primitives to applications for constructing protocols and registering them with active nodes. When defining a protocol the user makes a subclass of the class `Protocol` provided in the ANTS Toolkit which constructs the proper code groups and protocol structures to carry the capsule information that is needed in the network.

ANTS provides an API for the construction of Protocols, which includes the functions: *startProtocolDefn()* to indicate that a new protocol is being defined, *startGroupDefn()* that indicates the definition of a new code group for a particular type of capsule and *addCapsule(String name)* to add the new type of capsule to the code group and protocol structure. The new protocol is then registered with the defining node so that the corresponding capsules can be recognized for processing. Without the code group and the protocol definition the capsule is considered to be anonymous and, for security reasons, only default forwarding is performed on it.

Applications

Establishing the active nodes and the whole active networking environment requires the construction of `Applications` at the end-systems. ANTS provides the `Application` class to users who create their own custom applications by sub-classing it. The `Node` class exports an API to support applications accessing the node environment and communicating with it. Users create new applications to deploy new custom services at the active network nodes. It is through an application that an instance of a related protocol is created and registered with the necessary nodes. Once the proper definition of the protocol and code group is done, an application is allowed to send and receive capsules of the corresponding type from the node. The `Application` class provides methods including *send()* and *receive()* for communicating over active networks using newly defined protocols.

2.8.3 Security Issues and Resource Management in ANTS

Active Networks become more prone to intrusion by untrusted users and malicious code because they provide a facility for users to customize the network. ANTS provides access

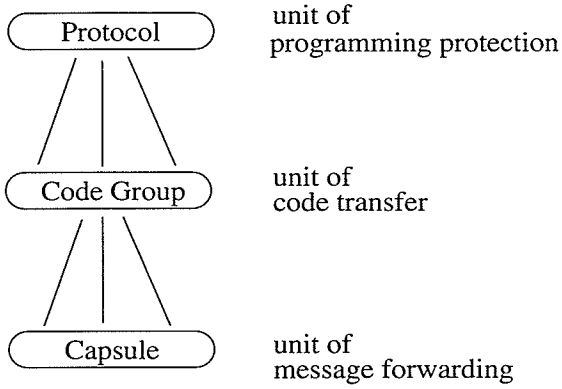

Figure 2.8: Capsule Composition Hierarchy (Adopted from [Wet98])

to all users whether trusted or untrusted and gives them the authority to handle their own capsules. The risk increases when such authority is given to users, as code written by a user might interfere with the processing of another user's code or could adversely affect the whole network infrastructure. The protection model that ANTS uses does not allow such malicious behavior by packets in the network. The type field in the capsule provides security by clearly specifying the processing routine to be used to handle the capsule. It is generated using an MD5 [Riv92] cryptographic fingerprint of the associated code. Every capsule has a specific fingerprint associated with it, which clearly specifies the forwarding routine, the code group and protocol they belong to. This approach makes it very difficult to cause an active node to execute unexpected code. Furthermore, hacking a capsule to create a capsule of another type is prevented since ANTS strictly restricts capsules from changing their types. This is enforced by processing capsules belonging to different services in different sand box environments. Fingerprints are only created at the edge nodes when capsules are injected into the network. They can not be manipulated by active nodes inside the network. The threat of any type of service interfering with another type of service is thus handled in ANTS.

Other security-related threats such as the corruption of the Node environment, corrupt

service code, and manipulation of the state cached by one type of capsule by another type are also handled in ANTS. Nodes use very safe evaluation techniques for demultiplexing capsules to their specified service code as just described. Further, the ANTS environment exploits the security properties of the Java language to ensure node security.

After forwarding a capsule, the node stores any corresponding state in its soft-store which is partitioned by protocol type. This prevents stored capsule state from being accessed or manipulated by capsules of a type which is not a part of the storing capsule's protocol. As shown in Figure 2.8 the protocol is the unit of protection so for two forwarding routines to share data they must come from the same protocol.

The management of network resources to avoid inappropriate over consumption must also be supported in any network environment. ANTS provides resource management techniques that scale efficiently in active networks environments. The most common concerns are the processing of a single capsule consuming a large amount of resources at some node, as well as creating other capsules which may consume resources at other nodes (e.g. network flooding). Another possible problem is an end-system sending a large number of capsules and congesting the network. ANTS handles the use of network resources by capsules, service code and applications through continuous monitoring of the node and network runtime environments. A watch dog utility incorporated into the ANTS Toolkit does not allow forwarding routines to run for an overly long time and terminates the processing followed by deleting the cached state associated with the terminated routine to ensure future performance. Capsule processing that might consume an unduly large amount of memory and network resources is precluded by the "resource limit" field of each capsule. The resource limit field in the capsule is used to break any routing loop the capsule may have got stuck into in a fashion similar to the use of the TTL field in conventional IP packets.

Chapter 3

Problem Definition

The problem addressed in this thesis is to examine the possibility of using in-network "resource discovery" to dynamically construct non-dedicated clusters from collections of workstations that are predicted to be underused. This chapter provides some background information concerning the problem, discusses how the problem might be solved using active networks, and then discusses some assumptions made about the environment in which such a solution might be employed.

3.1 Problem Background

Solving many important research problems requires a large amount of computing power. Both universities and industries are investing a lot in buying parallel computers and in building dedicated clusters to meet these computational requirements. A more cost-effective solution to this problem, however, is to use computational resources that are already available but which are under-used.

Pfister [Pfi98] identifies three ways of improving performance:

- Work harder,
- Work smarter, and
- Get help.

Working harder, technologically, means working with higher performance processors and devices. Working smarter refers to finding more efficient ways of solving compute intensive problems and getting help, in this context, suggests collecting together several machines to use in solving a single compute intensive problem.

Huge pools of under-used compute cycles are normally available in many organizations because most individual's workstations are not used 24 hours a day. The only way to use these cycles, however, is to find and organize them. This research aims to solve one of the problems involved in creating clusters using these under-used computational resources. Specifically, it deals with the problem of finding the available resources, a problem that is often referred to as "resource discovery"[1]. The goal of this resource discovery is to be able to dynamically create clusters that will be suitable for use in solving long-running, high performance computing problems.

3.1.1 High Performance Computing

Buying a commercial high performance system to solve an HPC problem is an alternative that is not always feasible due to the cost involved. Even a small system with eight processors, several Gigabytes of memory and sufficient storage space may cost more than many ordinary organizations or researchers are willing to pay to solve their compute

[1]Resource Discovery is the process of locating available machines in the network and determining useful related information such as their load characteristics as well as the characteristics of the network(s) connecting them.

intensive problem(s). While cheaper, dedicating a collection of uni-processor computers to do the same compute intensive job may still not be cost-feasible for some researchers.

Processor technology is improving so fast that the power of a single-chip CPU is comparable to the processors used in supercomputers only a few years earlier. Intel Pentium series processors (among others) have surprisingly high processing power and are widely used around the world. When such high power processors are made available to everyone throughout both large and small organizations (where the jobs performed are seldom compute intensive) the power of such processors is not fully utilized.

Over the past decade the perception of High Performance Computing has changed from high processing speed and performance on a single computer to high performance, processing speed and high communication speed over a distributed network of computers connected, typically, via a high speed "network" [Haw97]. Obviously, having a single computer with supercomputing power is a great asset for an organization but at the same time the idea of investing money in such a system seems inappropriate when many smaller, under-used computers throughout the organization are available. Many but not all HPC problems can be effectively solved using a "distributed computing" environment built from such existing machines.

Distributed computing, like parallel computing, works by dividing a job into several parts which are then assigned to multiple computers rather than multiple processors. Each computer performs its part of the job and returns the result, which is then combined to achieve the overall result. Obviously, getting a job done by several computers (as by several processors) working cooperatively has the potential to offer better performance than a single computer. As long as the cooperating machines in a distributed environment are interconnected by a sufficiently fast network, they can often be used nearly as efficiently as a parallel computer for many HPC problems. Further, the fault tolerance of distributed computing systems provides an added benefit. A super-computing system

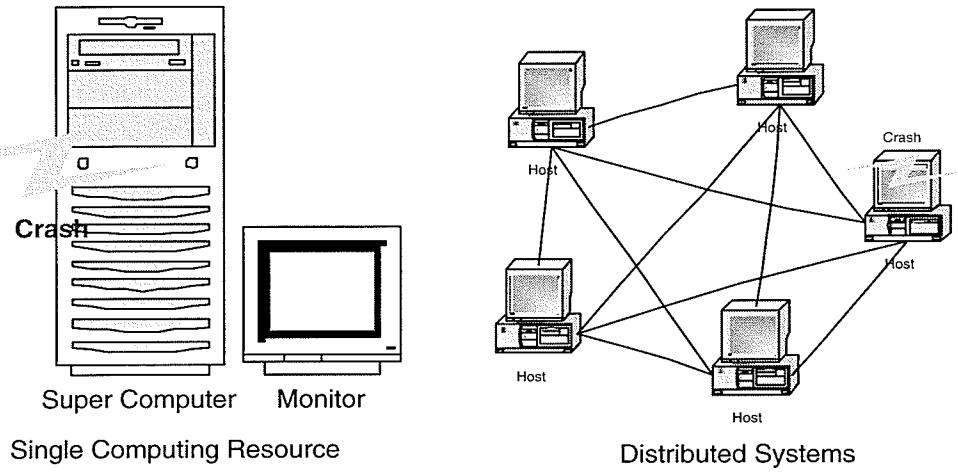

Figure 3.1: Super Computer vs Distributed System

having very high computing power and large memory is of no use if something goes wrong with the system since the whole machine may fail. In a distributed computing environment software can be written so that even if one computer fails the rest of the system will complete the job. Figure 3.1.1 compares the structure of a supercomputer and a distributed system.

3.1.2 Cluster Computing

In the 1960's IBM first introduced the idea of cluster computing by linking large mainframe computers to meet the needs of high end computing (without preventing mainframe users from accessing their existing applications). However, the development of "commodity" cluster computing had to wait until the development of high-speed networks, microprocessors with high performance capabilities and tools supporting distributed computing before it could be generally applied to solving problems in parallel [Raj99a].

Modern cluster computing is the collection of two or more interconnected machines brought together to create a distributed parallel computing system, which performs a single job in parallel much as a conventional parallel machine would. Clusters can be of two types, dedicated clusters such as Beowulf [Bec95] systems and non-dedicated clusters. The research described in this thesis deals with non-dedicated clusters. The difference between dedicated clusters and non-dedicated clusters is that in dedicated clusters the machines are used explicitly so that computing can be performed in parallel. Individuals do not own the machines that are a part of the cluster. Whenever an individual wants to perform a job, he/she just submits the job to the cluster of machines (typically via a "host" node) and the job is done in a parallel fashion using as many processors as necessary and the result is returned to the individual once the job is finished.

With non-dedicated clusters (typically in a LAN environment), machines are owned by individuals. The main idea behind non-dedicated cluster computing is to "steal" the idle CPU cycles that a machine has without violating the rights of the owner of the system. A serious concern in non-dedicated clusters is that when the machine is used as a part of the cluster, the individual who owns that machine might not be able to use the full computing power of the machine. To use non-dedicated clusters for long-running parallel jobs it must be possible to find a way to predict the likely availability of the machines so they can be used when they are needed without hindering the progress of the individuals who actually own them.

The chief benefit of cluster computing (particularly with non-dedicated clusters), though it is less efficient than computing on parallel machines, is its cost-effectiveness. (Other features of cluster computing include high performance, scalability, and availability [Raj99a].) The performance of cluster computing is high since the machines that collectively form the cluster are connected with each other via a high speed LAN and therefore jobs may be performed in parallel by several processors rather than one.

The communication provided between the machines for executing parallel jobs is critical and provides the underlying network transport for message passing systems such as MPI [Pac98, Gro99], HPVM [Chi97], and PVM [Gei94].

A question that always arises is, "What will happen if one machine in the cluster performing a particular job fails". Given appropriate software, the job can potentially be transferred to some other machine that is still available [2] and then that machine becomes a new part of the cluster. Hence, the cluster is highly available (and is also scalable to meet application needs). Particularly, with non-dedicated clusters, it is often easy to find available machines as there is a potentially huge pool of under-utilized machines and these can be assigned to new jobs or added to existing clusters as required.

Another issue, especially in non-dedicated clusters, is the use of homogeneous vs heterogeneous collections of machines. The available machines in a network may be of different architectures (e.g. x86 vs Sparc vs Power PC) and the host Operating System (OS) in a cluster can also be different from machine to machine (e.g. Linux, Solaris, NT, etc.). Building a cluster with different operating systems and/or architectures leads to a heterogeneous cluster while a collection of identical machines running the same operating system forms a homogeneous cluster. In this research heterogeneous clusters are not considered [3]. The focus of the thesis is to develop and run the code successfully on a homogeneous cluster to illustrate proof of concept. Heterogeneous clusters are considered to be future work.

[2] Assuming such a machine is available

[3] With the use of appropriate software (e.g. XDR, XML, etc.) integrated into the cluster programming software (e.g. MPI) heterogenous clusters could be used.

3.2 Problem and Solution Motivation

Several systems such as Condor [Lit88] and NWS [Wol97, Wol98, Wol99] have been developed in an effort to exploit computational resources in widely distributed systems. The approach used in such systems is centralized (which is prone to failures that can lead the whole system to fail). Condor for example, has a central coordinator, which manages the registry of all the workstations and is a central authority for scheduling the jobs onto the workstations (as described in Section 2.6). If the central coordinator fails then the whole Condor system stops working. Similarly, in the NWS system a single process may be a single point of failure. If a critical process such as the one that maintains the load information or the one that forecasts the future availability of machines fails, the whole system will be halted. Unlike Condor, NWS doesn't use any distribution where the process maintaining the information is concerned. In Condor, at least every workstation has its own job pool and scheduling authority. Only the registry information for the workstations and load measurements is maintained by the central coordinator. The central coordinator, makes the decision of which job's to be scheduled on which workstations but it doesn't have to worry about the actual scheduling of the jobs which is taken care of by the distributed workstations participating in the system which maintain their own job queues. This means that if the central coordinator fails briefly, work will still continue on each node so brief failures can be tolerated.

Overloading such centralized coordinators (and the network around them) is also a concern. For example, since a single process running on one machine in the NWS system maintains the load information received from all the participating systems, the machine running that process is prone to getting overloaded when the system scales to a large network of workstations. The load gathering processes sending the load measurements to the single process might overload the process and may also cause nearby network

congestion, thereby, significantly slowing down the whole system and possibly leading other critical processes to starve.

These problems with a centralized system lead to the idea of a distributed (in this thesis, hierarchical) system which is less prone to such failure conditions. To overcome the problem of a single failure point, an architecture needs to be developed which maintains the necessary information but is not centralized. Using a hierarchical architecture, the system can be divided into several levels with a superset of information at higher levels and a means of effective communication between them to disseminate information. Using a hierarchical structure, if a node at a certain level fails to be able to do something it can send the task to its upper level (parent) node which will have broader vision. More specifically, the nodes at each level can store the measurements received from various machines beneath them and can forward those measurements to their parent nodes. Thus, a node sending measurements to a higher node has the measurements of all the machines beneath it and the node at the level above has the measurements of these machines as well as the machines that lie under other nodes that are one level below it. Thus, the vision of a parent level node is broader than that of its children.

It should be noted that the term "Node" in the previous paragraph, need not be limited to host machines. A very efficient and robust implementation of such hierarchical resource discovery can be done in-network. Routers are at an ideal position for gathering the information on resource use and making decisions concerning what resources should be allocated to satisfy a cluster formation request. Unfortunately, conventional routers do not have the ability to store the measurements to enable cluster formation decisions to be made. As described in earlier chapters Active Networking facilitates the deployment of user code into the network which can provide such an ability.

Using Active Networking, the concept of hierarchical in-network resource discovery and management is possible. In this thesis, ANTS will be used to deploy customized

code to various active routers so as to form a resource discovery hierarchy. The nodes will run continuously and collect information through capsules which will be sent by monitoring daemons installed on the candidate host machines. These daemons will be configured to send load measurements into the network. The measurements received from the daemons will be stored at the active nodes in a data structure which can later be used for forecasting the availability of resources. After receiving and storing the measurements, each active node will forward the capsule to the active node one level above it in the hierarchy. The higher level active node in the hierarchy will thus receive such measurement capsules from all active nodes that lie under it in the hierarchy. Thus active nodes at upper levels will have a much broader vision of the resources that are available than active nodes at lower levels and will be able to satisfy correspondingly more aggressive allocation requests.

Once the active nodes have load measurements from various machines, a user can send a cluster formation request asking for certain resources by using a special purpose capsule. Such capsules can be recognized as "resource allocation requests" and, upon receiving such a capsule, an active node can call forecasting and cluster creation routines to satisfy the request. If not satisfied at a given level a request capsule can be forwarded to the next higher level, which of-course has a broader vision of the resources available (over a wider geographic area) to see if the request can be satisfied there. Only if the root level router can't satisfy a request does it fail.

3.3 Assumed Computing and Networking Environment

The "resource discovery" system just described works assuming a hierarchy. The daemons responsible for discovering the resources must continuously probe host-machines (end nodes) and send their load measurements into the network (to the active routers)

where they are then disseminated subsequently to other active routers higher in the hierarchy.

This assumed hierarchical structure need not be the actual structure provided by the available network. All that matters is that the parent child relationship be known to the nodes. Considering the arrangement of the nodes (active nodes and end nodes) as a connected graph, it is always possible to embed a tree structure into the existing network topology (as, for example, is done when constructing core based trees for multi-cast communications [Bal93]).

The intention in developing this system was to make it applicable in wide-area contexts. As will be shown in Chapter 6, scalability results of the system to a large number of machines over a wide area network are favorable. An obvious goal of the cluster formation system developed in this thesis is, of course, to have the largest possible pool of machines to choose from when doing cluster construction This will enable greater success in cluster formation as well as support for larger clusters but will require wide-area application. Given current network technology and characteristics, the system described in the following chapter should work well in a local-area environment with high-speed switched networks but some wide-area networks may be too slow.[4]

The prototype implementation described in this thesis runs in a local-area environment consisting of x86 based Linux machines available in different labs at the University of Manitoba. Advanced high-speed research WANs have been and are being deployed. CA*Net3, for example, is an optical Dense Wave Division Multiplexed (DWDM) network that is capable of carrying up to 40 Gbps of traffic[5]. The system described may

[4]If we build a cluster of nodes that must communicate over a slow network link the performance of the parallel applications will be poor.

[5]Interested readers can find more information on CA*Net3 at Canada's Advanced Internet Development Organization web page available at the URL- http://www.canarie.ca/advnet/canet3.html

work well in such a high-speed environment, although latency issues will still limit the range of parallel applications that may be effectively supported.

To be of practical use, the proposed system must be compatible with existing cluster programming facilities. The "resource discovery" system described will create a set of potentially allocatable nodes that can be used to form a cluster of the required size and capabilities at the requested time. Information on available machines will initially be stored in the soft-stores of active nodes together with their corresponding measurement information. A cluster construction (forecasting) routine will then select the nodes to form the cluster. The result of this selection will be used to construct a cluster configuration file, which will be used as input to a system such as MPI when the parallel job for which the cluster was requested is finally run. A batch system in between the "resource discovery"/forecasting system and MPI (or other parallel runtime systems) could be used to automate the process of running job(s).

Chapter 4

Problem Solution

The solution to the problem described in Chapter 3 is provided using three distinct components (and, correspondingly, three active protocols). The first component discovers the computational[1] resources available in the network. The second component supports user interaction with the system allowing users to request cluster resources and receive responses from the third component which deals with the management and processing of discovered resource information. The third component is divided into two sub-components, one which interacts with the first component to gather and store resource information and another which interacts with the second component to respond to user requests by identifying appropriate resources to be used to form clusters having the requested characteristics and returning that information to the user.

The system presented works assuming a hierarchical network structure and that all active nodes know their place in the hierarchy. While a hierarchy is assumed, the actual topological arrangement of the workstations in the network may be different. The

[1]The research presented in this thesis deals only with computational resources and therefore manipulates only such information as CPU load, memory capacity, etc. and not with network load issues such as bandwidth, latency, etc. Network load issues are considered to be future work.

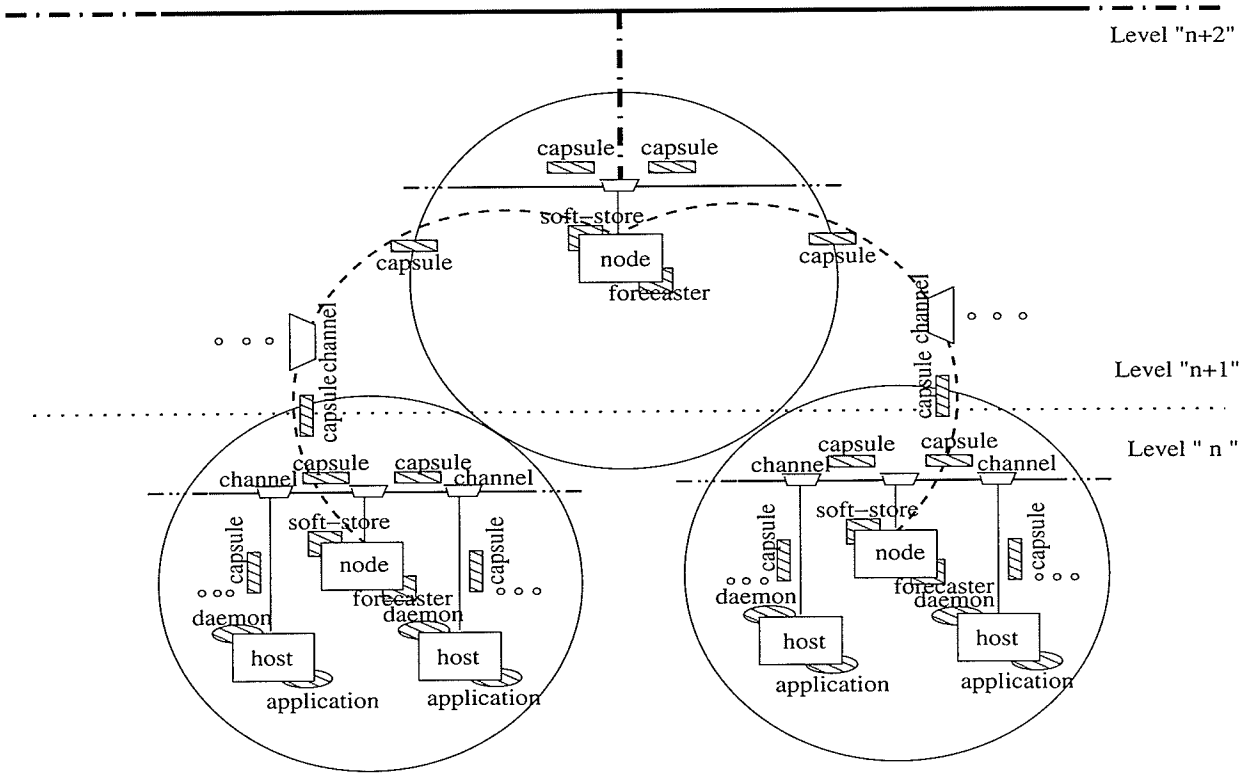

Figure 4.1: Assumed Hierarchical Architecture

workstations participating in the system may actually be connected to each other without any hierarchy among them, but the system will logically configure the workstations as a hierarchy (i.e. the system will embed a hierarchy in whatever topology actually exists).

The hierarchical arrangement of active nodes in the system is important since the hierarchical structure helps to overcome the limitations on scalability and fault tolerance in a centralized system. Given a hierarchical arrangement of nodes such as that shown in Figure 4.1, the parent-child relationship of the nodes can be used to facilitate architectural fault tolerance. If a node at one level, say level "n", fails then the whole system need not fail since a higher level node (at level "n+1") can assume the responsibilities of its subordinate node. How this is accomplished will be described later in the thesis.

4.1 Resource Discovery

As described earlier, a huge pool of computation resources is under-utilized in many organizations, and the only way to make use of them is to find and organize them. To do this, a daemon process is installed on each workstation that is used to gather load, etc. information that can later be used to form clusters. Each daemon will run periodically (perhaps every 5 or 10 mins) to make load measurements for the corresponding workstation (host). This code is referred to as a "Sensor" daemon since it senses the workstation load periodically by collecting certain OS-provided statistics. In the prototype implementation, there are three pieces of information that are fetched from the workstations; the current load of the CPU, the amount of free memory and the amount of disk space available. Collectively, this information is referred to as a <C,M,D>[2] triple (an example is shown in Figure 4.2). To get these measurements, the Sensor daemon uses the Linux */proc/cpuinfo* file to determine the total CPU capacity (expressed in Bogomips) and a Linux utility called *procinfo*[3] to fetch the percentage of idle CPU cycles, the free memory and disk space. The *procinfo* utility simply reads the */proc* directory to gather the needed system information and supply it in a nicely formatted fashion. The idle CPU percentage (expressed in Bogomips) is calculated later at each active node by multiplying the total CPU cycles available (in Bogomips) by the idle CPU cycles percentage.

The Sensor daemon also determines the day and time at which the probe was conducted. The daemon then encapsulates the load measurements and time information in a capsule called a "SensorCapsule" and injects it into the network anonymously[4]. The assumed hierarchical architecture is such that the workstations running these daemon processes and injecting capsules into the network lie under some specific active node,

[2] 'C' stands for "Compute", 'M' for "Memory", 'D' for "Disk".

[3] Interested readers can find more information in the Linux System Manual

[4] For example, by using a hardware supported broadcast capability.

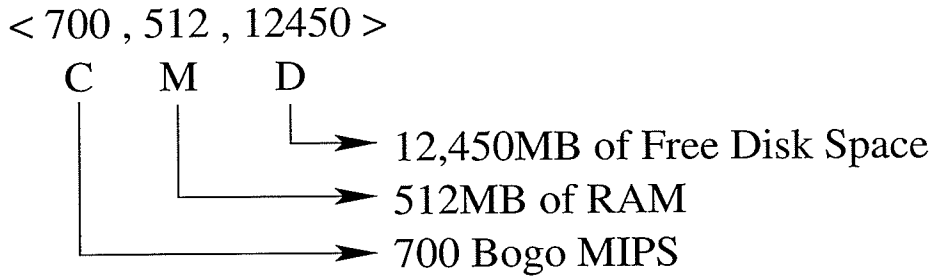

Figure 4.2: <C,M,D> Triple

which can capture the injected capsules and process them. (In case of the failure of an active component at the node immediately above the injecting host it is assumed that the packet will still be forwarded to the active node one level higher in the hierarchy which will capture the capsule for processing). Thus, the daemon processes don't need to know the actual address of any active node(s). This "anonymity" simplifies the process of configuring the system and simplifies the Sensor code generally. Further, the failure of the active processing at a single node does not stop the process of cluster formation.

4.2 Active Router Processing

The "Active Nodes" (routers) are the main component in the system being described. The routers are at an ideal position in the network to gather information about potentially under-utilized resources and to make forecasts concerning the availability of resources. This is because the assumed hierarchical network structure places each router at the root of a "tree" of hosts from which clusters may be constructed. Routers at higher levels have knowledge of larger groups of hosts and can be used to satisfy requests to form large clusters. Clusters will always be formed at the lowest possible level and this will normally result in clusters with the fastest possible interconnection between the hosts

(since they will be physically closer together).

Each active node will capture the capsules injected by daemon processes lying beneath it using the active code installed and executing on it. This code is capable of recognizing the Sensor capsules and processing them. The active code specified in each Sensor Capsule will result in the contained (payload) information being stored at the active node for future use in selecting hosts to use to form clusters.

4.2.1 Measurement Storage

"Sensor" capsules received by the active nodes contain measurements of the workstations <C,M,D> values which need to be stored at the active nodes so that they can be used later for forecasting the availability of resources. Each active node uses a circular queue based data structure in its soft-store for storing the measurements received from the Sensor daemons. The information stored in the queue is tagged with the machine identifier (IP Address) where, and day and time at which, the measurement was taken and will also contain the <C,M,D> information itself. Measurements for a relatively long period (e.g. a week) will be accumulated and stored in the soft-store since repeated access patterns can be expected over such a period and these patterns can be used to predict future load. The information stored in the soft-store is illustrated in Figure 4.3.

Once a Sensor capsule has been processed and the measurements are stored in the soft-store of the active node, the capsule must then be forwarded to the next higher level active node. Since the nodes are arranged in a hierarchical manner, forwarding the Sensor capsules received to upper levels of the hierarchy makes the resources described "available" to the higher level nodes. This increases the number of hosts visible to the active nodes as we travel up the hierarchy. Thus, a service request that cannot be satisfied at one level due to scarcity of resources available, can simply be sent on to a higher level,

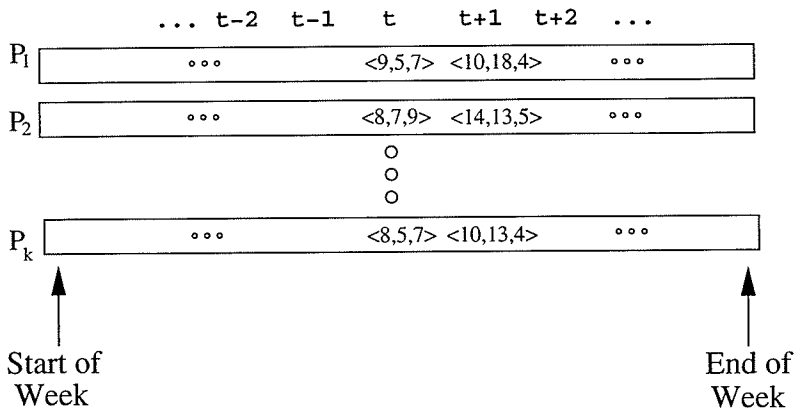

Figure 4.3: Measurement Storage Pattern in Soft-Store

which will have access to a larger number of resources. All but the largest of potential requests should be serviceable in this way.

4.2.2 Service Provision

The second sub-component involved in active router processing deals with service provision. Since the measurement information is stored in the router's soft-stores, an active node can easily access the information to provide cluster creation services for the host machines under it. Such a host machine needs a protocol to request cluster creation service from the active node, and both the active nodes and the host machines must agree on this protocol. Active nodes will receive "Request" capsules from hosts for this purpose. Such a "Request" capsule will contain the number of requested resources, duration of time for which the resources are requested, and the day and time for which the request is made. In response to receiving such a Request capsule, an active node will invoke code to match the requirements specified in the request with the measurements stored in its soft-store to make a forecast about the availability of the needed resources. If adequate

resources are known to the router then the request is served by forming a cluster of machines that are expected (based on past behavior) to be available at the required time and the list of selected machines is returned to the requesting host via a "Response" capsule. If the request cannot be fulfilled, then the active node simply passes the request to its parent (the active node at the level above it) which has stored information about more machines. The same process is then followed, if necessary, at the next level and so on. The formation of a cluster thus takes place dynamically which is atypical in conventional cluster computing environments. The list of machines returned to the requester can then be used to construct a job-specific MPI (or similar) hosts file which can then be used at the appropriate time to run the MPI (or similar) job.

When an allocation is made, it is desirable to pass information about the allocated resources[5] up and down the hierarchy in the same way that measurements are disseminated. This will avoid the re-use of allocated workstations for constructing clusters for the same (or for an overlapping) time duration as when the workstation will be a part of another cluster. For this purpose every active node must also maintain a resource allocation list "RAList", which is checked prior to allocating resources. Every active node, after allocating resources to a request, creates a "ResourceUpdate" capsule which encapsulates a summary of the resources being allocated and sends it to its parent active node. This capsule also follows the same procedure as the "Sensor" capsule, in that the information is fetched and the capsule is forwarded to the next level and so on. The RALists maintained at each active node are updated with the information fetched from the ResourceUpdate capsules. The entries in an RAList store the time duration (start time and end time) for which the described resources have been allocated. This process of resource updating enables all the active nodes in the hierarchy to have the latest infor-

[5]Allocated resources here refers to those workstations, which have been chosen as a part of a cluster by an active node at a certain level of the hierarchy.

mation concerning the resources in use and the time duration for which they have been allocated. While processing a RequestCapsule, every active node first checks the RAList to get any information about the allocated resources for the requested day and time and these are considered while processing requests to allocate resources.

4.3 Request Processing

Some request processing software will be used to allow users to interact with the dynamic cluster creation system just described. A host/user machine will use this software to prompt the user for the needed cluster parameters (e.g. number of machines, capacities, etc.) and will then inject a service request into the network. Such a service request will take the form of a "Request" capsule that contains the requirements for a parallel job expressed as a sequence of triples for each required machine together with the length of time the machines are required. An active node receiving such a request capsule will execute the active code in the capsule to extract the request information. This request is then processed at the active node as described previously.

Chapter 5

Implementation

The ANTS Toolkit was used as the basis for the development of a prototype system that constructs clusters dynamically from idle workstations. ANTS is a Java based Toolkit that provides a capsule based Active Network Transport System for dynamically deploying Active Code in an active network. As a result the implementation is coded in Java, primarily as a set of ANTS classes.

The active-network-based prototype described in this chapter was deployed for testing purposes within the Department of Computer Science at the University of Manitoba on machines distributed in different labs as shown in Figure 5.1. The machines that were used for forming the clusters were all x86 based Linux machines[1]. The prototype exploits an assumed hierarchical network structure (as described in Chapter 3). The hierarchy of router's enables the easy dissemination of load information in a useful and efficient manner. As described earlier, every router has knowledge of the load measurements of all the machines and routers that are beneath it. Thus, a service request that cannot be met at

[1]The test environment consisted only of Linux machines since only a Homogeneous environment was considered in the prototype.

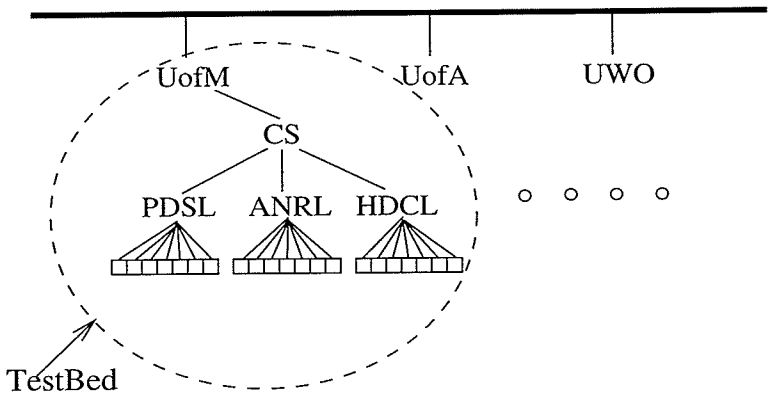

Figure 5.1: Test Bed

one level (due to lack of available resources or code failure) can then be sent to the level above.

Recall that the active processing that must be done can be divided into "Resource Discovery" (done at the candidate cluster nodes), and "Service Provision" (requested by a user and provided by some active node). The implementation will be discussed in terms of these separate processes.

5.1 Resource Discovery

As described, a daemon process is installed on all the host nodes of the distributed system. The host nodes acquire certain exported API's from the ANTS environment but they do not have any active code installed on them. (They are aware of the new capsule types and can recognize and create capsules as needed.) For example, the daemon process uses the exported API's that support a user application sending and receiving capsules which it inherits from the Application class of the ANTS Toolkit. Locally defined (as sub-classes of the Protocol class) and registered protocols for each type of capsule (as shown in Algorithm 1) allow the nodes to create the necessary capsules which will even-

Algorithm 1 Daemon Process Algorithm (Registering Protocols and Collecting Stats)

```
1  public class Sensor extends Application implements Runnable
2  String file = "tmp";                          /* temporary file to store probe results */
3  public void start() throws Exception
4  getNode().register(newSensorProtocol());      /* register protocols */
5  getNode().register(newResponseProtocol());
6  getNode().register(newRequestProtocol());
7  getNode().register(newResourceUpdateProtocol());
8  P = writeToFile(file);
9  fm = read file("/proc/cpuinfo");              /* read /proc/cpuinfo to find CPU speed */
10 while (!eof) do
11       P.write(fm); od
12 /* capture output of the "procinfo" command to find other host info */
13 Process q = runCommand("procinfo");
14 stdInput = captureOutput(q);
15 while (!eof) do
16       P.write(stdInput); od
17 P.close();                                     continued in Algorithm 2...
```

tually pass through the active nodes which will process the capsule forwarding routines to implement the cluster creation process.

5.1.1 Daemon/Sensor Process

The processing done by the Daemon process begins with probing the workstation on which the Daemon is installed (as shown in Figure 4.1 in Chapter 4) to capture its load

information. The daemon process reads the */proc/cpuinfo* file (provided by Linux) to determine the approximate "speed" of the CPU in BOGOMIPS. The reading of the file is shown in Algorithm 1 (lines 9 through 11). The information read is written to a temporary file "tmp" (line 2 of the algorithm). This file is purely for convenience and is later removed once the probe is completed. After having read the */proc/cpuinfo* file, the daemon process runs the Linux "procinfo" command to get the current load information for the workstation and the output of this command execution is also written to the "tmp" file as shown in Algorithm 1. The daemon also runs a "date" command and writes its output to the "tmp" file although this is not shown. Once the "tmp" file is written the daemon uses a file input utility to read tokens from the "tmp" file, as shown in Algorithm 2. The *LineListTokenizer1()* utility is adapted from the ANTS Toolkit and has been modified to extract the required data from the "tmp" file. The information that is needed from the "tmp" file is the date ("day"), the amount of memory ("memory"), the actual CPU capacity ("total CPU") and the amount of idle CPU time ("idle CPU"). *LineListTokenizer1* reads the file token-wise and extracts the required measurement information conditionally by checking every Token read.

The measurement values extracted are stored in separate variables and an instance of *SensorCapsule* is then created to encapsulate the measurement parameters as shown in Line 20 and 21 of Algorithm 2. *getNode().time()* gets the local time and puts a time stamp in the capsule to specify the time when the capsule was created. *day, Time, bogo, free, idle, ipA* denote the day on which the measurement wass taken, the time on that particular day, the available CPU speed in bogomips, the free memory, the percentage of idle CPU and the IP address of the machine probed, respectively. Two *port* fields specify the source and destination ports, which are ANTS specific, and *target* is the active node address to which the capsule is to be sent. The capsule is then injected into the network at Line 22.

Algorithm 2 Daemon Process Algorithm (Processing Stats and Sending Capsule)

```
1  LineListTokenizer1 lt = new LineListTokenizer1(file);
2  String value = lt.retval();
3  for (i = 0; i < 5; i++) do          /* There are five parameters to extract */
4      switch(i)
5      case 0:
6      day = Integer.parseInt(value[1]);
8      case 1:
9      Time = value[i];
11     case 2:
12     bogo = (int)Float.parseFloat(value[1].trim());
14     case 3:
15     free = (int)Integer.parseInt(value.trim());
17     case 4:
18     idle = (int)Float.parseFloat(value[i].trim()); od
19 String ipA = nsLook();              /* calls a method to fetch IP Address of host */
20 SensorCapsule cap = new SensorCapsule(getNode().time(), day , Time , bogo,
21 free, idle, ipA, port, port, target, null);
22 send(cap);                         /* injects capsule into the network to be received by router */
23 wait for 15 minutes before conducting next probe
```

5.1.2 SensorCapsule Format

The general format of a *SensorCapsule* is shown in Figure 5.2.

The measurements contained in a *SensorCapsule* are :

⋆ *timestamp*: Time at which the probe was conducted.

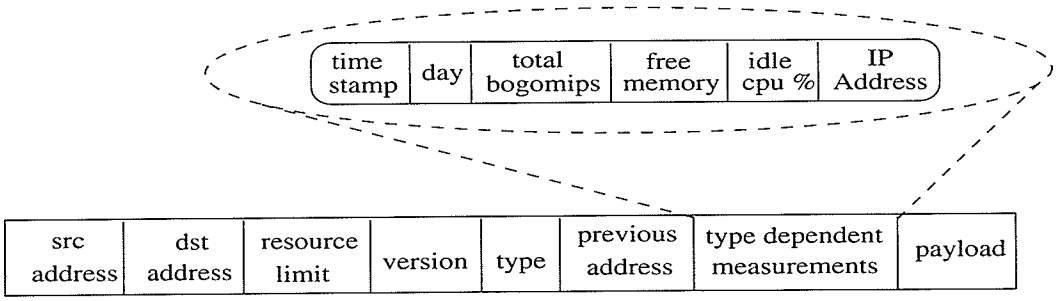

Figure 5.2: SensorCapsule Format

- ⋆ *day*: Day of the week that the probe was conducted (since in the prototype, the measurements are only stored for a single week).
- ⋆ *total bogomips*: Total CPU capacity available in Bogomips.
- ⋆ *free memory*: Amount of free memory available.
- ⋆ *idle cpu time*: Idle CPU time expressed as a percentage.
- ⋆ *IP Address*: IP Address of the workstation on which the probe was conducted

Each *SensorCapsule* also inherits a number of fields (shown in the bottom of Figure 5.2) from the *Capsule* class.

5.1.3 Sensor Protocol

The *Protocol* that defines the capsule format and the corresponding code group that provides the active code for processing sensor capsules is shown in Algorithm 3.

The Protocol definition defines a unique Code Group for the capsule at line 4 in Algorithm 3 and then adds the capsule to that code group at line 5. All the instances of *SensorCapsule*, therefore belong to the same code group. The unique code group

Algorithm 3 Algorithm for the SensorProtocol

```
1 public class SensorProtocol extends Protocol
2 public SensorProtocol() throws Exception
3 startProtocolDefn();
4 startGroupDefn();
5 addCapsule("mycode.SensorCapsule");
6 endGroupDefn();
7 endProtocolDefn();
```

defined allows active nodes to identify *SensorCapsules* and their associated code during code distribution.

5.2 Active Router Processing

Active router processing is the main component of the system described in this thesis. The application *NodeApplication* manages the active processing of different capsules to maintain the measurement information and later make forecasts of workstations to form clusters. The *NodeApplication* class is divided into two parts as described earlier, which provide measurement storage and service provision, respectively. The active *NodeApplication* (NA) runs continuously as an application level router at each level in the network hierarchy to capture the capsules being injected by the *Daemon* processes.

5.2.1 Measurement Storage

To have the necessary information available for the construction of clusters, the NA creates the needed queue data structure in the Soft-Store as shown in line 3 of Algorithm 4.

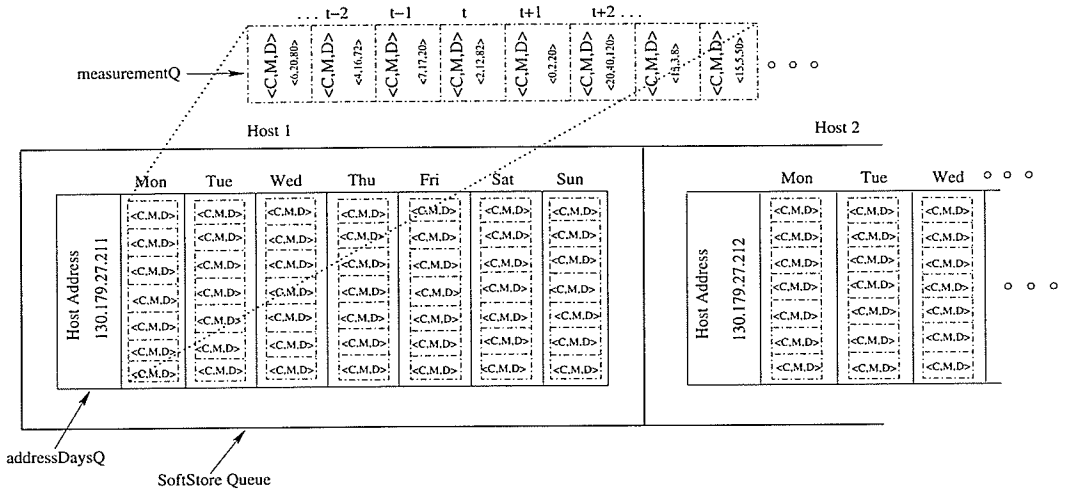

Figure 5.3: Node Soft-Store

The soft-store is a three level queue structure used to store captured load measurements from multiple hosts over an extended period (1 week in the prototype). Figure 5.3 shows the structure of the Soft-Store managed by the NA. The three level nature of the queue structure is evident in the diagram. The third level queue *measurementQ* is embedded in the queue *addressDaysQ* which is itself embedded in the queue *SoftStore*. The Soft-Store queue maintains statistics for one workstation per entry ("column" in Figure 5.3) as depicted by the "Host1", and "Host2" tags in Figure 5.3. The *addressDaysQ*s maintain the IP address of the workstation for which the measurements are stored and a collection of *measurementQ*s. In the prototype, the *measurementQ* is shown as seven columns that store the measurement statistics for a week's period. Every *measurementQ* is further segmented into 96 time slots for storing the <C,M,D> triples collected in one day (since the Daemon process probes the workstations every 15 minutes). Dividing a week by 15 minutes (the scheduling interval) and multiplying by 7 days results in a sequence of 672 <C,M,D> triples for each workstation [2]. The NA code calculates the

[2] Assuming 4 bytes per field, this gives a storage requirement of $12 * 672 = 8064$ bytes per workstation in each active node's soft-store

appropriate slot in the *measurementQ* where each <C,M,D> triple is to be stored based on the time the sample was taken (eg. midnight Monday). Once a week is over (i.e. after Sunday at 23:59) the measurements received are stored again starting with the measurements for Monday overwriting the previous data. Such a storage system provides the NA with the latest measurement statistics to use while constructing a cluster. The NA code calls a slot computation routine every time it needs to store or retrieve a load measurement. For example, when a *SensorCapsule* is received, the *TimeStamp* is supplied as a parameter to the slot computation routine. The hours, minutes and seconds are fetched from the *TimeStamp* and the time is converted to seconds (from midnight). Since the prototype probing interval is 15 minutes, the interval becomes 900 sec as shown in Equation 5.1. The time in seconds is calculated using Equation 5.2 and then the slot is determined using Equation 5.3.

$$interval = 15 * 60 = 900sec \tag{5.1}$$

$$time = (hours * 60 * 60) + (minutes * 60) + (seconds) \tag{5.2}$$

$$slot = \frac{time}{interval} \tag{5.3}$$

The NA code also maintains an *RAList* as shown in line 4 of Algorithm 4, which is the list of resources already allocated. The *RAList* is checked each time the NA code processes an allocation request to prevent the system from allocating already allocated resources.

The NA code captures the capsules in Algorithm 4 at line 7 and checks the type of the capsule received in Algorithm 5. The process of protection (finger print based) is a

Algorithm 4 Algorithm for the Active Node Application

```
1 public class NodeApplication extends Application implements Runnable
2 int target;
3 SoftStore softVec = new SoftStore();          /* creates a Soft-Store */
4 RAList raList = new RAList();          /* creates a Resource Allocation List */
5 addressDaysQ addressDVec;
6 measurementQ measurementQVec;
7 synchronized public void receive(Capsule cap)
8 super.receive(cap);                          continued in Algorithm 5
```

built in feature in ANTS, the NA code only has to check the type of the capsule received and process it according to the corresponding code as shown in Algorithm 5. Capsules of type *SensorCapsule* are forwarded to code in the NA that checks to see if the Soft-Store is empty. If the Soft-Store is empty it creates a new storage queue for the corresponding capsule (received from a particular host). Otherwise, it de-multiplexes the measurements to their corresponding queue by matching the IP address in the queue with the one in the capsule. If the measurements are received from any host for the first time the NA code creates a new queue for that host.

Once the measurements are stored in the Soft-Store the NA code sets the destination address of the capsule to be the address of its parent (higher level) router and executes a forwarding routine that forwards the capsule up in the hierarchy, thereby disseminating the measurement information upwards as shown in Lines 20 and 21 of Algorithm 5.

Algorithm 5 SensorCapsule Processing Algorithm

```
1  if cap instanceof SensorCapsule
2    if (softVec Empty)                          /* check if the soft-store is empty */
3      create(addressDVec) and create(measurementQVec)
4      softVec.add(addressDVec);/* soft-store is empty create all queues for this entry */
5      addressDVec.store(cap.IPAddress);
6      addressDVec.add(measurementQVec);
7      measurementQVec.atColumn(cap.Day).add(cap);
8  else
9      Check cap.IPAddress == softVec.addressDVec(getIPAddress())/* soft-store not empty */
10     if exist(cap.IPAddress)                     /* IPAddress exists in soft-store */
11       measurementQVec.append(cap); /* direct the capsule to respective queue */
12   else
13       softVec.add(addressDVec);          /* capsule from host received first time */
14       addressDVec.store(cap.IPAddress);/* create new queue for first time arrivals */
15       addressDVec.add(measurementQVec);
16       measurementQVec.atColumn(cap.Day).add(cap);
17     fi
18   fi
19 fi
20 cap.setDst(target);                          /* "target" is address of above active node */
21 cap.evaluate(getNode());
```

5.2.2 Service Provision

Service provision deals with providing services to users wishing to have clusters constructed to solve their intensive computational problems. This part of the system makes

use of the measurements stored by the active nodes for the construction of potential clusters by providing a processing routine for handling cluster creation requests received from potential users in *RequestCapsules*. A *RequestCapsule* (described in the next section), when received by an NA, is demultiplexed to the corresponding code that will handle the construction of the necessary cluster based on the specifications fetched from the *RequestCapsule*. Recall that these specifications include the number of processors needed, when and for how long they are needed, and the <C,M,D> values for each requested processor.

The NA code for service provision first checks the *RAList* created when the NA starts for the first time (Algorithm 4, line 4) to see if it has resources available for the requested time period. If the *RAList* is empty then all resources are potentially available for allocation. Otherwise, the NA code checks the already allocated resources for the requested time period described in the *RAList* and makes a list of suitable processors as shown in Algorithm 6 at lines 6 through 11. The *canAllocate* and *requestSatisfied* flags are set accordingly based on the availability of resources. If the *canAllocate* flag is false the *RequestCapsule* is simply forwarded to a higher level active node. Otherwise, a list of allocatable resources is created as shown in line 7 of Algorithm 8 after matching the resources in the allocated resource list, with a record of how many resources are available in the *numProc* counter as shown in line 4 Algorithm 8. If the number of resources requested is equal to the number of resources in the allocatable resource list as shown in line 11 then a *ResourceUpdateCapsule* is created (lines 14 and 15), and sent to the upper level node so it can update its allocated resources list. At the same time a *ResponseCapsule* is created, line 17 and sent to the requesting host announcing that the specified resources are at its disposal for the requested time period.

The format of a *ResponseCapsule* is shown in Figure 5.4 and contains:

Algorithm 6 Algorithm for Creating an Allocated Resource List

```
1  /* This Algorithm checks to see whether there are resources available to allocate */
2   /* if so then it creates an allocated resource list and forwards it to Algorithm 8 */
3  if cap instanceof RequestCapsule
4    if raList.size() != 0
5      for raList.size() do
6          if RequestTime == AlreadyAllocatedTime
7            if NumberOfResources(raList) == NumberOfResources(softVec)
8              canAllocate = false;              /* there are NO resources available */
9              requestSatisfied = false;
10         else      /* create list of allocated resources (to be used in Algorithm 8) */
11             create AllocatedResourceList(AllocatedResources)
12             canAllocate = true;
13             requestSatisfied = true;
14           fi
15           canAllocate = true;
16           requestSatisfied = true;
17         fi
18     od
19 else
20     canAllocate = true;
21     requestSatisfied = true;
22   fi                                          continued in Algorithm 8...
```

⋆ *timestamp*: time at which the capsule was created.

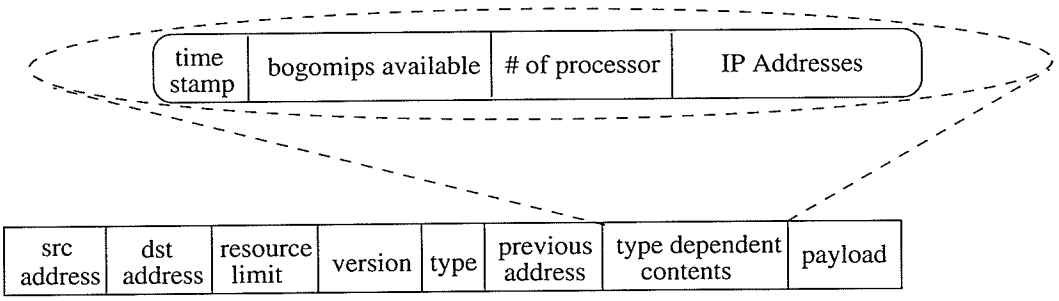

Figure 5.4: ResponseCapsule Format

Algorithm 7 Algorithm for the ResponseProtocol

```
1 public class ResponseProtocol extends Protocol
2 public ResponseProtocol() throws Exception
3 startProtocolDefn();          /* ANTS Protocol class provides these functions */
4 startGroupDefn();
5 addCapsule("mycode.ResponseCapsule");
6 endGroupDefn();
7 endProtocolDefn();
```

- ⋆ *bogomips available*: CPU cycles at the requesters disposal.
- ⋆ *no. of processors*: number of processors at disposal.
- ⋆ *IP Addresses*: IP Addresses of the workstation of the cluster.

Again, the *ResponseCapsule* class inherits the standard capsule fields from its base class *Capsule*.

The capsule types *ResourceUpdate* capsule and *ResponseCapsule* are registered with each active node via the protocols *ResourceUpdateProtocol* and *ResponseProtocol* shown in Algorithms 10 and 7 respectively.

Algorithm 8 Resource Allocation Algorithm

```
1  if canAllocate == true                          /* continued from Algorithm 6 */
2    while int i < softVec.size() and numProc < Rcap.ProcR/* until enough resources */
3          for int l = 0 ; l < allocatedAddresses.size(); l++/* is the resource allocated ?*/
4              if softVec.resource(i).getSrc() == allocatedResources(l).getSrc()
5                 resourcesAllocated = true;
6              else                                /* if not allocated then "allocatable" */
7                 allocatebleResources[numProc] = softVec.resource(i).getSrc();
8                 numProc++;
9              fi od
10         i++; od
11   if numProc != Rcap.NumProc        /* do we have required no. of processors */
12     resuestSatisfied = false;
13 else                              /* send ResourceUpdateCapsule and ResponseCapsule */
14     ResourceUpdateCapsule RUCap = new ResourceUpdateCapsule(getNode().time(),
15     Rcap.ProcR, Slot, checkToSlot, day, Addresses, port, port, target, null);
16     send(RUCap);
17     ResponseCapsule Recap = new ResponseCapsule(getNode().time(), availableBogo,
18     numProc, Addresses1, port, port, cap.getSrc(), null);
19     send(Recap);
20   fi
xelse
22   if requestSatisfied = false or canAllocate == false
23     cap.setDst(target);                          /* forward it to above node */
24     cap.evaluate(getNode());
25   fi
26 fi
```

5.2.3 Processing of ResourceUpdate Capsules

The NA also provides a routine for processing *ResourceUpdate* capsules. When the NA code receives a *ResourceUpdate* capsule, it checks its type and de-multiplexes it to the corresponding routine.

Algorithm 9 Algorithm for Processing ResourceUpdate Capsules

```
1 if cap instanceof ResourceUpdateCapsule
2   ResourceUpdateCapsule ruCap = (ResourceUpdateCapsule) cap;
3   Resource resource = new Resource(ruCap.numProc, ruCap.fromTime, ruCap.toTime,
4    ruCap.day);
5   System.arraycopy(ruCap.address, 0, resource.address, 0, ruCap.numProc);
6   raList.add(resource);
7   cap.setDst(target);
8   cap.evaluate(getNode());
9 fi
```

Algorithm 9 shows the processing of a *ResourceUpdate* capsule when it is received by an active node. The data received from the *ResourceUpdate* capsule is stored in an *RAList* as an instance of the *Resource* class, as shown in line 3 of Algorithm 9. The field *resource.address* is an array of IP addresses of the allocated resources, which is fetched from the *ruCap.address* array. An array containing all the addresses of the resources allocated is also encapsulated in a *ResourceUpdate* capsule and is disseminated to the active nodes in the hierarchy (both up and down). Once the *RAList* is updated, the capsule is forwarded as shown in lines 7 and 8 of the algorithm.

The format of a *ResourceUpdate* capsule is shown in Figure 5.5 (including the inherited fields), and contains:

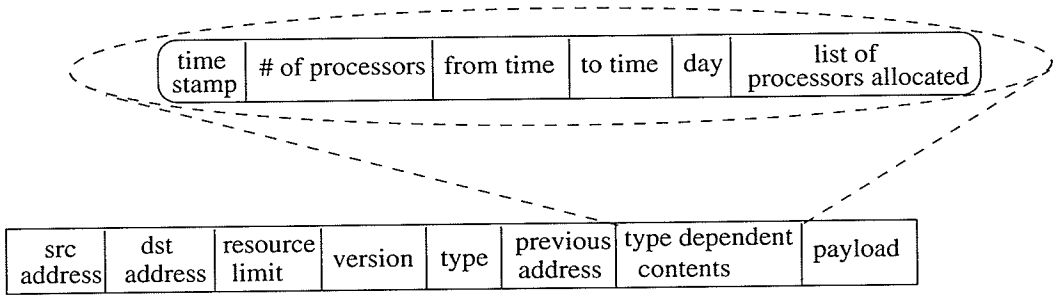

Figure 5.5: ResourceUpdateCapsule Format

* *timestamp*: time at which the capsule was created.
* *no. of processors*: number of processors that have been allocated.
* *from time*: starting time from which the resources are allocated.
* *to time*: end time, when the resources will be freed.
* *day*: day for which the resources are allocated[3].
* *list of processors allocated*: IP Addresses of the workstations that have been allocated to form the cluster.

The *ResourceUpdate* protocol is shown in Algorithm 10. The corresponding *ResourceUpdate* capsule is strictly defined to belong to its own group and protocol as shown in lines 3 through 7.

The *Resource* class which defines a single unit of data stored in an *RAList* (used at line 3 in Algorithm 9) is shown in Algorithm 11. The variable *address* is an array that stores ANTS specific addresses of the machines and *ipaddress* is an array that stores the corresponding IP addresses of the machines that are allocated. *fromTime*, *toTime* and *day*

[3]The prototype doesn't allow requests for multiple days

Algorithm 10 Algorithm for the ResourceUpdateProtocol

```
1 public class ResourceUpdateProtocol extends Protocol
2 public ResourceUpdateProtocol() throws Exception
3 startProtocolDefn();          /* ANTS Protocol class provides these functions */
4 startGroupDefn();
5 addCapsule("mycode.ResourceUpdateCapsule");
6 endGroupDefn();
7 endProtocolDefn();
```

Algorithm 11 Resource Class Initialization

```
1  public class Resource
2  int[] address;
3  int fromTime;
4  int toTime;
5  int day;
6  String[] ipaddress;
7  public Resource(int pr, int ft, int tt, int da)
8  begin
9      address = new int[pr];
10     ipaddress = new String[pr];
11     fromTime = ft;
12     toTime = tt;
13     day = da;
14 end
```

specify the start time from which the machines participate in the cluster, when they will be freed and on what day the machines have been allocated, respectively.

5.3 Request Processing

A potential user wanting to execute a compute intensive job on several processors will send a request to the active nodes by using the *RequestApplication*. The *RequestApplication* in the prototype simply provides a GUI to allow users to enter the requests and their parameters which are then sent to the nearest active node by pressing a "send" button.

The method *actionPerformed(ActionEvent evt)* at line 1 of Algorithm 12 listens for the button pressed event and fetches the data (i.e. cluster parameters) entered on the form. It creates a *RequestCapsule* using this data and sends it to the nearest active node as shown in lines 11 and 13. This capsule is processed by the service provision code of the active node that receives it and the results are sent back to the *Request* application announcing the resources that are available for its use. This is accomplished using a *ResponseCapsule* which is received by the *RequestApplication*. The *RequestApplication* then fetches the IP addresses from the *ResponseCapsule* and uses them to construct a host file (listing the cluster hosts to be used for a given job) for the MPI (or other) code that will run the job on these machines.

To recognize the capsules *RequestCapsule* and *ResponseCapsule*, the *RequestApplication* registers the protocols as shown in Lines 2 and 3 of Algorithm 13.

5.3.1 Request Capsule Format

The general format of a *RequestCapsule* is shown in Figure 5.6 and has following fields:

Algorithm 12 Algorithm for the RequestApplication

```
1  public void actionPerformed (ActionEvent evt)
2  if evt.getSource() == sendRequest
3     new Thread(this).start(); fi
4  public void run()
5  String timeForm = Time.getText();
6  String dayForm = Day.getText();
7  int hourForm = Integer.parseInt(HoursRequired.getText());
8  int bogo = Integer.parseInt(Bogo.getText());
9  int mem = Integer.parseInt(Mem.getText());
10 int proc = Integer.parseInt(Proc.getText());
11 RequestCapsule cap = new RequestCapsule(getNode().time(), timeForm, dayForm,
12 hourForm, bogo, mem, proc, port, port, target, null);
13 send(cap);
15 synchronized public void receive(Capsule cap)
17 if cap instanceof ResponseCapsule
18    ResponseCapsule reCap = (ResponseCapsule) cap;
19    for reCap.address.length
20          System.out.println("Machines at Disposal- "+reCap.address[j]);
21          constructHostFileMPI(reCap.address[j]);/* constructs MPI host file */  od
22 fi
```

- *timestamp*: time at which the capsule was created.
- *from time*: starting time from which the resources are allocated.
- *to time*: end time, when the resources will be freed.

Algorithm 13 Protocol Registration and GUI

```
1 public start()
2 getNode().register(new RequestProtocol());
3 getNode().register(new ResponseProtocol());
4 A simple GUI created using Java AWT
```

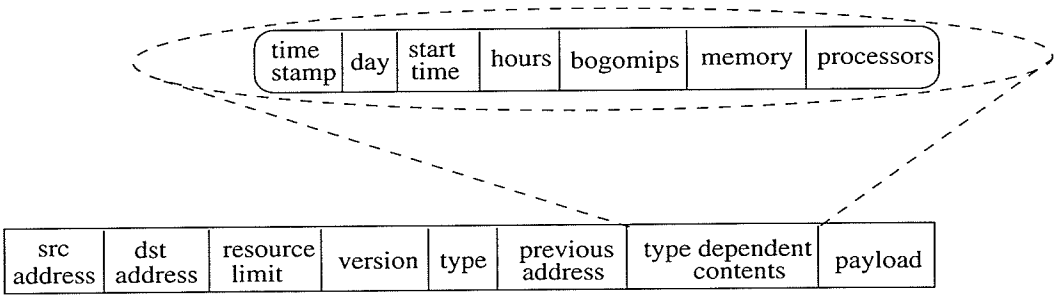

Figure 5.6: RequestCapsule Format

⋆ *day*: day for which the resources are allocated.

⋆ *Processors*: Number of resources allocated.

5.3.2 Request Protocol

The definition of the protocol used between an active node and the requesting application is shown in Algorithm 14.

Algorithm 14 Algorithm for the RequestProtocol

```
1 public class RequestProtocol extends Protocol
2 public RequestProtocol() throws Exception
3 startProtocolDefn();          /* ANTS Protocol class provides these functions */
4 startGroupDefn();
5 addCapsule("mycode.RequestCapsule");
6 endGroupDefn();
7 endProtocolDefn();
```

Chapter 6

System Assessment

In this chapter, I will present arguments that suggest that the proposed system is, in fact, useful. This will be done by considering the potential scalability of the system and the cost of using it. No simulation system was developed since there is no comparable system to evaluate it against. The benefits offered by the technique proposed in this thesis (improved fault tolerance, anonymity, and distribution of workload) are clear and simulation is not necessary to illustrate them. The existence of the prototype code serves to illustrate that such a system can be built. Its exact performance will depend on implementation details that are outside the scope of this thesis.

6.1 System Scalability

The prototype implemented in this thesis runs over a network of computers available in various labs in the Computer Science Department of the University of Manitoba. The test-bed environment consists of systems based on 550 MHz Intel Pentium III processors (which translates to 1094.45 bogomips) and running the Linux operating system (RedHat

No. of Machines	Search Time (in milliseconds)
10 machines	less than 1 ms
100 machines	approx. 8 ms
1000 machines	approx. 54 ms

Table 6.1: Running Time to Search Soft-Store

7.2 version). In the testbed, the active code was run on relatively few systems. A possible concern is whether or not the system is scalable. That is, whether or not such a system, as it is implemented in the prototype, could run efficiently if the network scales to include many machines. The chief issue here is the requirements placed on the active nodes so the relevant question to ask is if an active node at some level in the hierarchy can maintain and process information about 100, 1000, 10000, etc. machines [1].

To handle large-scale networks, active nodes must be able to store relatively large amounts of host load information in their soft-stores. This is unlikely to seriously impact active routers near the edge of the network (since they will support relatively few hosts beneath them) but will be an issue for active routers higher up in the hierarchy (which must store much more load information). Fortunately, the capacity of memories are continuing to increase rapidly and the costs are very low. This means that it is reasonable to assume that storage capacity is unlikely to be an issue in active router design.[2]

It must also be possible for such higher level active routers to search their soft stores to respond to resource allocation requests efficiently. To get an idea of the efficiency of

[1] Environments with hundreds or a few thousand machines could easily exist within a single organization (e.g. a University) where relatively high-speed interconnects can be assumed and sharing of machines would be relatively easy so it is important that it be possible to handle this many processors.

[2] Additionally, many commercial routers now support the use of an "attached" processor for providing greater computational capability so, if necessary, such storage could be offloaded from the router core.

the active code developed for the ANTS environment a simple timing test (code shown in Algorithm 15) for searching the soft store was done. The test simply runs the active code (in an ANTS active node) on a single machine, initializes its soft-store to store the dummy measurements for 10, 100, and 1000 machines, in turn and then, once the soft-store is initialized processes a request capsule sent to the active node with a request to form a cluster of up to 10, 100 or 1000 processors, respectively for a given time period. The time taken to find the requested processors is determined primarily by the time taken to search the whole soft-store (i.e. every queue is traversed in an unoptimized fashion). The respective running times for the 10, 100 and 1000 processor cases are shown in Table 6.1.

Algorithm 15 Timing Code for Testing Scalability

```
1 lastTime = System.currentTimeMillis();   fetches the system time in milliseconds
2 begin
3   Code for searching the soft-store...
4   ...
5   ...
6 end
7 time = (System.currentTimeMillis() - lastTime);
8                                          current time minus the last fetched time
```

The results show reasonable performance in searching the soft store even for 1000 machines. For very large distributed systems, the search time may be non-negligible but this is ameliorated by two factors. First, the response time for cluster construction is not particularly critical. If a user submits a cluster creation request and does not receive a response for several seconds, this would still be perfectly acceptable. (The time frame for running such compute intensive jobs is, at a minimum, on the order of hours and

may be many days or even weeks so a few second delay is not a problem.) Second, a production implementation of the proposed system would likely use a more efficient active networks platform than ANTS (e.g. PAN [Nyg99]). Even something as simple as the use of a Just-In-Time (JIT) compiler for the Java-based ANTS code could provide a performance increase of an order of magnitude.

The times required to search the soft store have been shown to be small. Similarly, the processing required to update the soft stores will be small since update requests come infrequently (every 15 minutes in the prototype and likely not much more frequently in a "production" implementation). The only issue with respect to overloading the active routers may be at the highest level routers which will receive many more update requests than the lower level routers. This can be dealt with by providing a more powerful processing environment at the higher level routers (e.g. an "attached" processor) or, perhaps, by summarizing information from routers at one level before passing it up to higher level routers – an area of possible future work.

A related issue is the overhead introduced on the host machines by the sensor/daemon process that runs there. In the prototype, the processing performed is very minimal so the sensor has very low overhead. Even in a more complex implementation, it is hard to envision a sensor that would do sufficient computing to impact seriously on the performance of the machine it is running on. The presence of the sensor code should be transparent to users of the workstations involved in the system.

A final potential area of concern with respect to the feasibility of the proposed system is the issue of sharing. In a LAN environment, such as that used in the prototype implementation, secure access is freely available (due to distributed authenticated user ids) and the sensor code is normally easily accessible (through the use of a shared distributed file system such as NFS). In a wide-area implementation, such conveniences are typically unavailable. To address these concerns, each potential host machine will have

to provide an account that can be used to run non-local jobs as well as the sensor process. Setting up such an account and obtaining the sensor code would be a part of a necessary pre-configuration process for all machines participating in the system. Of course, once this initial step is complete, subsequent updates to the sensor code, etc. can be handled automatically as is currently done in volunteer computing systems (e.g. SETI@home, United Devices, etc.). For truly broad application, it may be desirable to implement the sensor and remote code execution via carefully controlled mobile agent code [Gra95] or using a controlled scheduling environment such as that proposed for Resource Containers [BDM99].

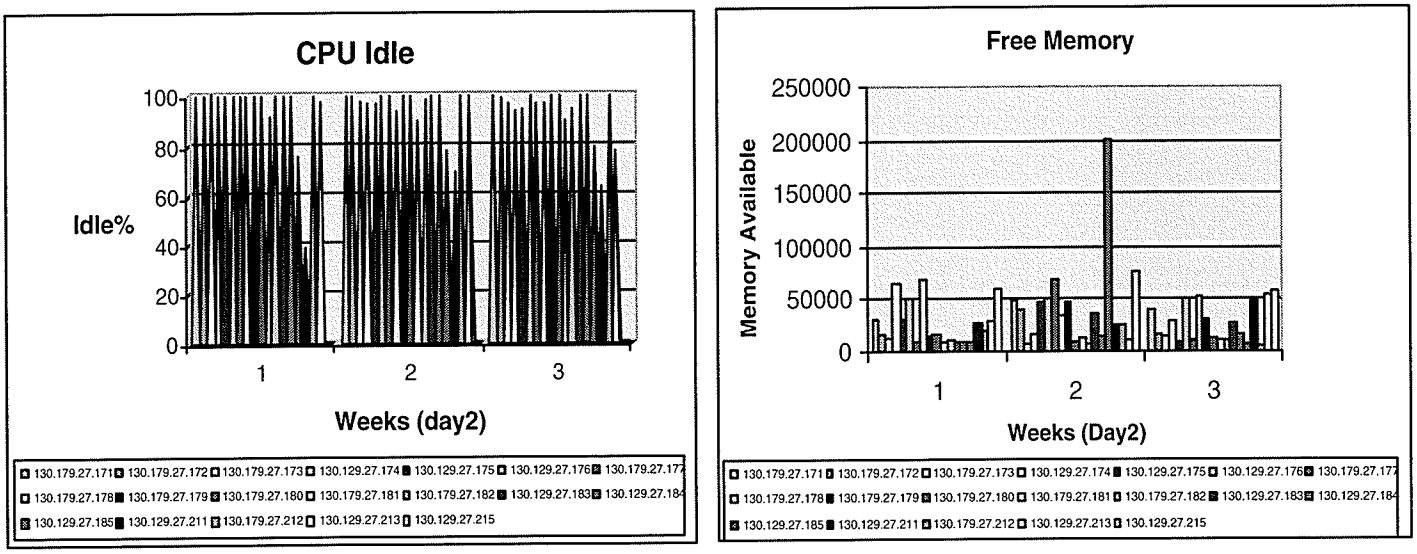

Figure 6.1: Statistics on Monday for 3 Weeks

6.2 Resource Availability

To get an estimate of how many machines might be available for processing heavy compute jobs, a simple test was conducted on the machines available in various labs within

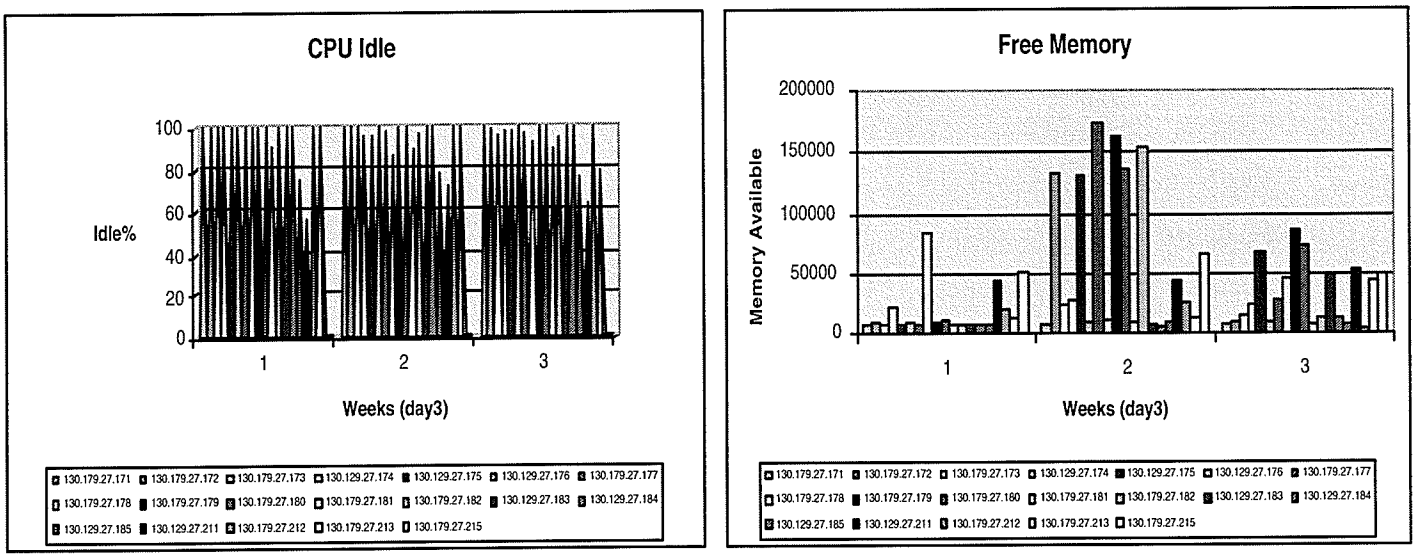

Figure 6.2: Statistics on Tuesday for 3 Weeks

the Department of Computer Science at the University of Manitoba. Several machines were regularly probled and the resulting statistics collected for a period of almost one month. Figure 6.1 shows the statistics collected from machines on Mondays over a 3 week period. It is clear that most of the machines are not fully utilized in terms of CPU cycles. Further, there are a few machines that have large amounts of memory available. Such machines could certainly be used by the system presented in this thesis to satisfy cluster formation requests.

Similar results are shown in Figures 6.2 through 6.7 for the other days of the week over the 3 week period. Examination of all the Figures shows that there is only one machine (IP Address 130.179.27.215), which is heavily loaded. In all the graphs the CPU utilization of this particular machine (IP Address 130.179.27.215) is maximum (only 0.3 percent on average of the total CPU cycles available are free).[3] This single machine would be the only one not to be selected to become part of a cluster during the

[3]The load on this machine was due to a single long-running simulation.

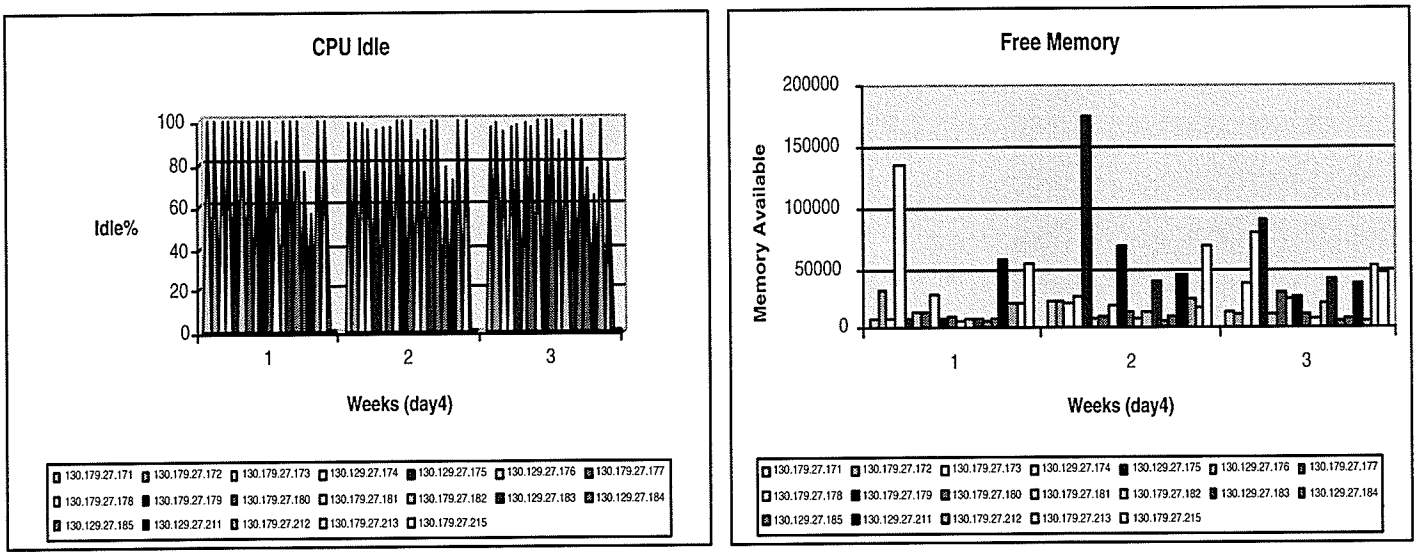

Figure 6.3: Statistics on Wednesday for 3 Weeks

three week period.

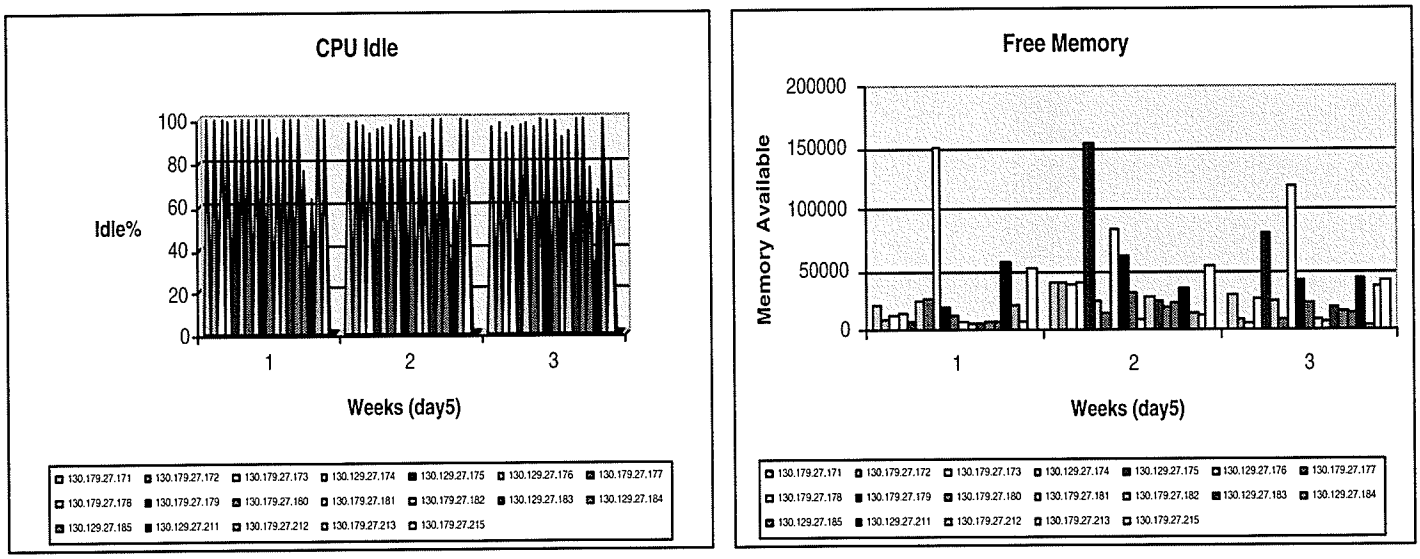

Figure 6.4: Statistics on Thursday for 3 Weeks

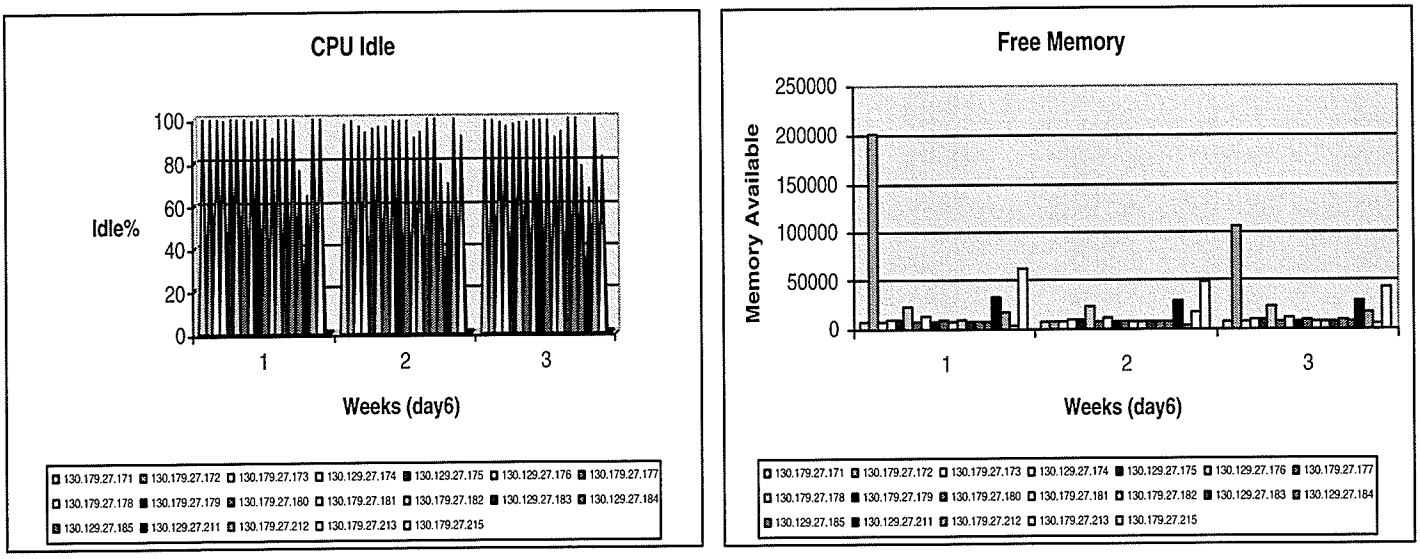

Figure 6.5: Statistics on Friday for 3 Weeks

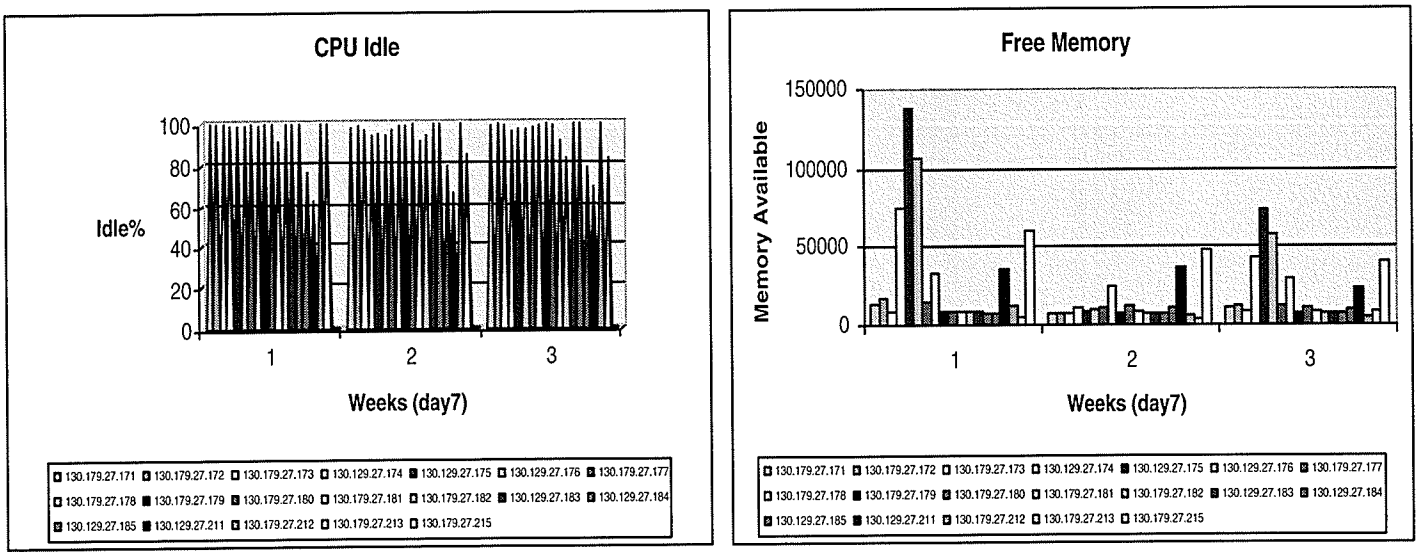

Figure 6.6: Statistics on Saturday for 3 Weeks

We can clearly see in Figure 6.8 that most of the machines have close to 100 percent of CPU cycles *Idle*. These machines can be used when constructing a cluster.

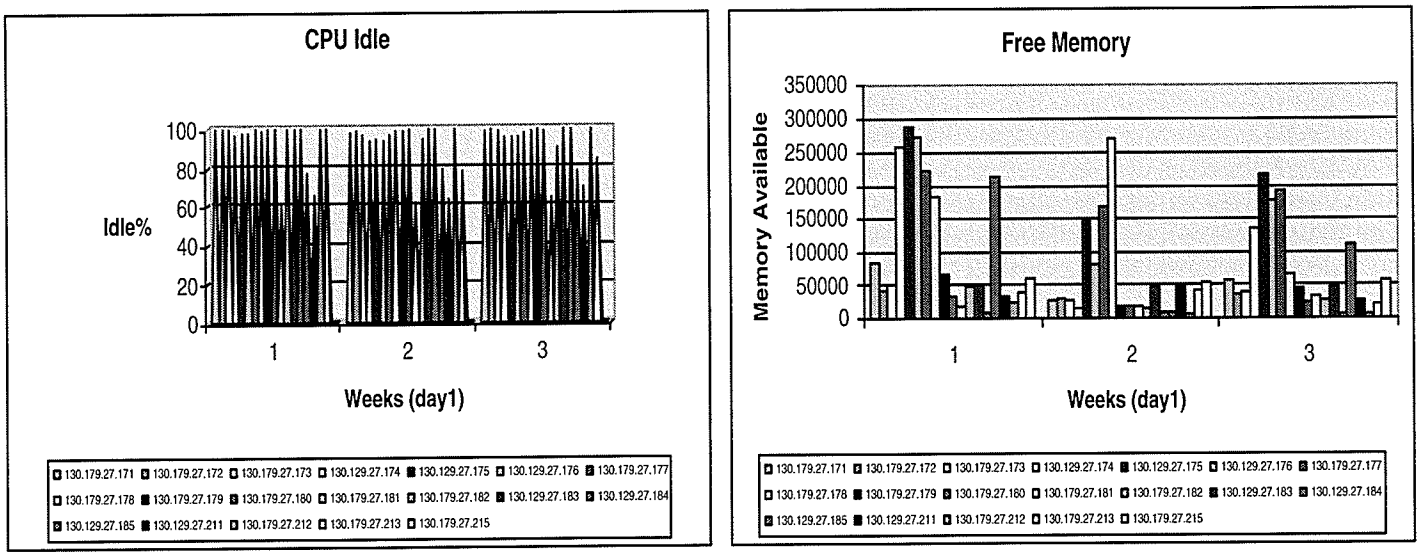

Figure 6.7: Statistics on Sunday for 3 Weeks

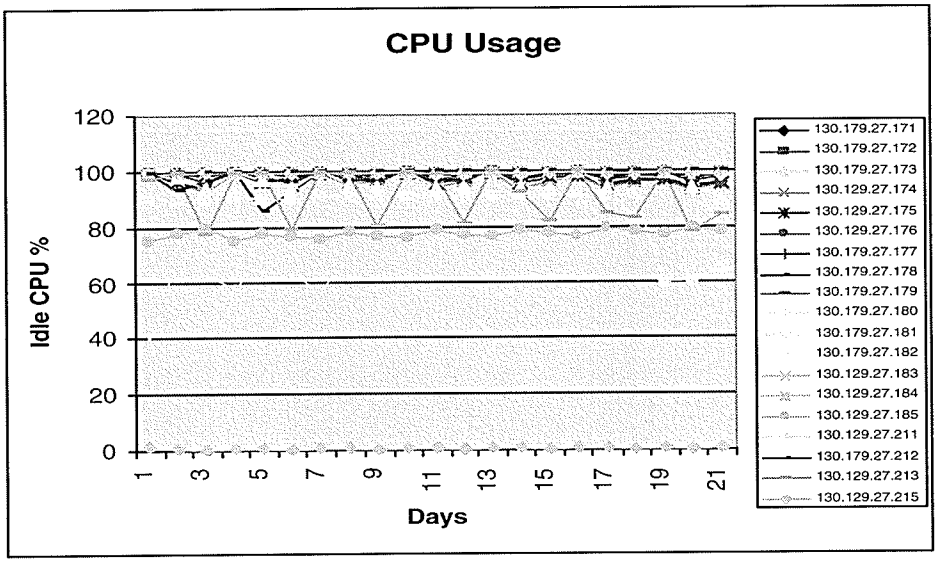

Figure 6.8: CPU Statistics for 21 days

Figure 6.9 shows the statistics of memory usage on different machines over the 21 day period. Clearly, there are several machines available with significant amounts of

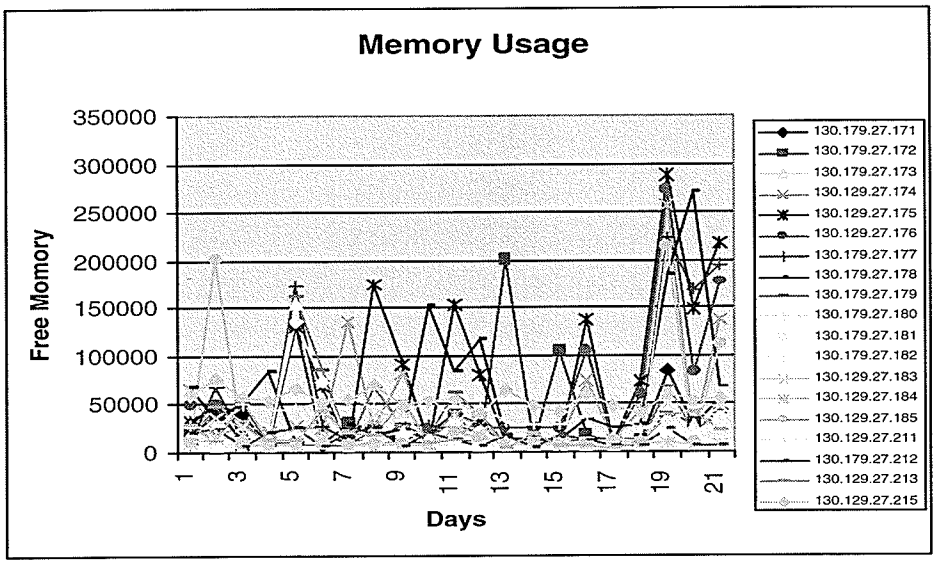

Figure 6.9: Memory Statistics for 21 days

free memory as well. These machines will be discovered and used to construct clusters requiring large amounts of memory.

It is interesting to note as well that the machines monitored over the 21 day period would be expected to be the *most heavily* used since they are in labs where most of the work is done. Private machines in individuals' offices should be even less heavily used. This suggests that the original premise of the thesis (that there are many unused computing resources available) is in fact correct.

Chapter 7

Conclusions and Future Work

The research presented is, to the best of my knowledge, entirely unique so there is no direct system with which to compare it. In essence, the presented system is in some ways similar to the NWS except that it runs in-network (using Active Networks as an enabling technology) to attain the benefits discussed earlier (anonymity, fault tolerance, and improved workload balance during resource discovery). The overall purpose of the system is quite different from that of NWS. Its goal is to enable the easy and efficient construction of (potentially) wide-area cluster systems "on-demand" using general computing resources that are predicted to be lightly loaded. In this respect, the system is similar to Condor [Lit88, Epe96] but it is more general in that it is not focused only on high-*throughput* computing. Additionally, unlike Condor, processes will not migrate once they are placed on specific nodes.

A proof-of-concept prototype was developed using the ANTS active network system to illustrate the feasibility of the technique and it was tested on a collection of X86 based Linux machines within the Department of Computer Science. Finally, some general arguments were made as to the practicality of the approach.

7.1 Future Work

There are many opportunities to extend and, in some cases, improve on the work presented in this thesis. The following are some issues that might be considered for future work:

7.1.1 Relaxing Constraints

A number of constraints were assumed in the thesis. Many of these may actually be relaxed to provide enhanced functionality relative to the prototype discussed.

Assumed Hierarchical network Structure

The presented research worked on the assumption that the active nodes (routers) were already arranged in a hierarchical fashion. An algorithm is needed to discover the actual network topology and embed a hierarchy in it. This can be done using a "flooding algorithm" whereby each router, in order to determine the available links to other routers will broadcasts a "Link State Update packet". Any routers that have a direct link receive the packet and send an acknowledgment back to the point of origin of the packet. By receiving the acknowledgment packets the routers determine the topology and can then agree upon a reasonable hierarchy to use. Ensuring that the flooding is done efficiently and determining and appropriate embedding are interesting problems to be solved.

Enhanced Computation Model

The load information gathered in the prototype (i.e. <C,M,D> triples) needs to be extended. The simple computation model works but has some significant limitations. Better

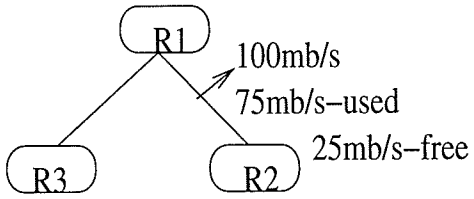

Figure 7.1: Network Speed

models could likely be created by including other significant load characteristics. An entire family of computation models could be developed that can be selected for use based, for example, on the characteristics of the cluster creation request.

Include a Network Traffic Model

The research presented deals only with load on the host machines. As a result, the routers use only those stored measurements in their forecasting algorithms. Load on the network was not considered in the research. Since the performance of parallel programs is highly dependent on the communication speed between processors, the predicted network load should also be considered.

Two important factors that should be included in a network traffic model are:

- ⋆ ***Network Interconnection Capacity :*** The network interconnection capacity on each link is obviously an essential consideration in any network traffic model. The interconnection capacity places a hard limit on the amount of data that may be exchanged in a given time period over the related link. Once a cluster is created and the jobs are assigned to the cluster additional network capacity cannot be added. The communication requirements of each job must be matched to the available network link capacity. Network interconnection capacity can be gathered at the same time that the "flooding algorithm" is run to determine the hierarchical embedding.

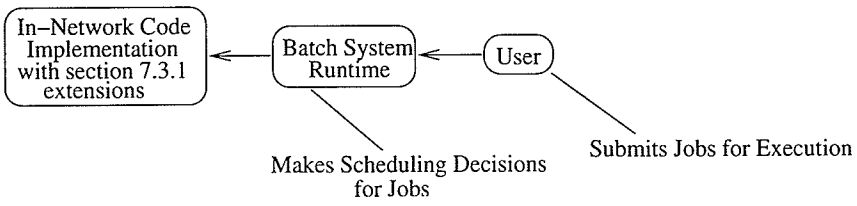

Figure 7.2: MPI Batch System Runtime

⋆ ***Job Communication Characteristics :*** The maximum available interconnection capacity alone is insufficient. The available capacity is also determined by the current traffic on the link(s) in question. For example, (refer to Figure 7.1) if the interconnection capacity of the network between two routers, R1 and R2, is 100 Mb/s and we know that currently 75 Mb/s is in use then only 25 Mb/s is free. In this case, a job that requires more than 25 Mb/s of transmission capacity should not be scheduled to machines that will use the interconnection between R1 and R2 for communication. A network traffic model should be able to predict such a situation using either pre-declared application communication characteristics or feedback information from previous job runs. It can use this information to form a cluster of machines where the available network speed is sufficient to meet the needs of the related job.

To track the network traffic, a new active protocol could be implemented that will gather the needed network usage statistics. The gathered information could then be stored in the routers soft-store the same way <C,M,D> triple from the host machines are stored. The active cluster formation code would then be extended to also consider the network traffic measurements in the forecasting process as well.

7.1.2 Possible Extensions

The following are some extensions that need to be done to make the presented technique practical to use.

Integration with Parallel Programming Tools

A new execution environment needs to be developed for MPI (or a similar parallel programming tool) that will use the information gathered by the routers to provide a cluster definition for use by the `mpirun` command, probably in a batch like environment. As shown in Figure 7.2. users who want to run an MPI job will submit their job's specifications to the batch system runtime environment (and specify an MPI program to be executed). The batch system runtime will communicate with the nearest active node to request a cluster meeting the needs of the job. Once a set of processors are returned, the scheduling system will take care of starting the MPI job on the selected machines at the appropriate time. This will simplify the job submission process for the users.

Alternatives to ANTS

The Active Networks Toolkit (ANTS) was used to implement the prototype system presented in this thesis. This results in certain performance limitations that may be a concern. As discussed earlier, these concerns can be simply dealt with if a performance alternative to ANTS (such as PAN) is used. It might also be possible to consider the use of a more general in-network computation environment. The use of ANTS in the prototype was expedient but not ideal. Much of the work done in-network to support dynamic cluster formation does not fit the ANTS capsule model particularly well. An interesting possibility would be to develop a new programming model where code could be executed

either in-network or on host machines using something like Mobile Agents[Gra95]. It has been observed that active network code and mobile agents share many properties in common and this suggests that agents (which provide a more general computing capability than active code does) might also be useful in-network.

Bibliography

[Ale97] Alexander D. Scott, Braden Bob, and Gunter Carl A. Active Network Encapsulation Protocol (ANEP). *Active Networks Group, RFC DRAFT*, 1997.

[Ale98] Alexander D. Scott, Arbough W., Keromytis A., and Smith J. A Secure Active Network Environment Architecture: Realisation in Switchware. *IEEE Network, 12(3):37-46*, 1998.

[And95] Anderson T., Culler D. and Patterson D. A Case for NOW. In *Proceedings of IEEE Micro, 15(1):54-64*, January 1995.

[Arb97] Arbaug W. A., Faber D. J., and Smith J. M. A Secure and Reliable Bootstrap Architecture. In *Proceedings of 1997 IEEE Symposium on Security and Privacy*, pages 65–71, May 1997.

[Bal93] Ballardie Tony, Francis Paul, and Crowcroft Jon. Core Based Trees (CBT) - An Architecture for Scalable Inter-Domain Multicast Routing . In *Proceedings of ACM SIGCOMM'93*, pages 85–95, 1993.

[BDM99] Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul. Resource containers: A new facility for resource management in server systems. In *Operating Systems Design and Implementation*, pages 45–58, 1999.

[Bec95] Becker Donald J., Sterling Thomas, Savarese Daniel, Dorband John E., Ranawak Udaya A., and Packer Charles V. Beowulf: A Parallel Workstation for Scientific Computation. In *Proceedings of the International Conference on Parallel Processing, CRC Press, Boca Raton, FL, USA*, August 1995.

[Bho98] Bhoedjang R. A. F., Ruhl T. and Bal H. E. Efficient Multicast on Myrinet Using Link-Level Flow Control. In *Proceedings of International Conference on Parallel Processing, Minneapolis, MN*, pages 381–390, August 1998.

[BM98] Wilkinson Barry and Allen C. Michael. *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers.* Prentice Hall, 1998.

[Bri92] Bricker Allan, Litzkow Michael and Livny Miron. Condor Technical Summary, Version 4.1b. Technical report, Computer Science Department, University of Wisconsin-Madison, Wisconsin, USA, 1992.

[Cal98] Calvert K., Bhattacharjee S., Zegura E., and Sterbenz J. Directions in Active Networks. *IEEE Communications, Volume 36*, pages 72–78, 1998.

[Cas94] Castagnera K., Cheng D., Fatoohi R., et al. Clustered Workstations and their Potential Role as High Speed Computer Processors. Technical report, NAS Computational Services RNS-94-003, NAS Systems Division, NASA Ames research Center, April 1994.

[Cas96] Casanova H. and Dongarra J. NetSolve: A network Server for Solving Computational Science problems. In *Proceedings of Supercomputing'96, Pittsburgh*, 1996.

[Chi97] Chien A., Pakin S., Lauria M., Buchanan M., Hane K. and Giannini L. High Performance Virtual Machines(HPVM): Clusters with Supercomputing APIs and Performance. In *Proceedings of 8^{th} SIAM Conference on Parallel Processing for Scientific Computing, Minneapolis, Minnesota*, March 1997.

[Epe96] Epema D. H. J., Livny M., Dantzig R. Van, Evers X. and Pruyne J. A Worldwide Flock of Condors: Load Sharing among Workstation Clusters. *Future Generation Computer Systems*, Volume 12, 1996.

[Fos97] Foster I. and Kesselman C. A Metacomputing Infrastructure Toolkit. *International Journal of Supercomputing Applications,11(2):115-128*, 1997.

[Fos99] Foster Ian and Kesselman Carl. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1999.

[Fos01] Foster Ian, Kesselman Carl and Tuecke Steven. The Anatomy of Grid: Enabling Scalable Virtual Organizations. In *Proceedings of Intl J. Supercomputer Applications, 15(3)*, 2001.

[Gei94] Geist Al, Beguelin Adam, Dongarra Jack, Jiang Weicheng, Manchek Robert and Sunderam Vaidy. *PVM Parallel Virtual Machine - A User's Guide and Tutorial*. MIT Press, 1994.

[Gra95] Robert S. Gray. Agent Tcl: A transportable agent system. In *Proceedings of the CIKM Workshop on Intelligent Information Agents, Fourth International Conference on Information and Knowledge Management (CIKM 95), Baltimore, Maryland*, December 1995.

[Gra01] Graham Peter and Singh Rajendra. A Mechanism for the Dynamic Construction of Clusters Using Active Networks. In *Proceedings of the 30^{th} Annual International Conference on Parallel Processing, Sept. 3-7*, 2001.

[Gri94] Grimshaw Andrew S., Wulf William A., Frence James C., Weaver Alfred C., and Reynolds, Paul F. Jr. Legion: The Next Logical Step Towards a Nationwide Virtual Computer. *Technical Report 94-21, Dept. of Computer Science, University of Virginia*, 1994.

[Gro99] Gropp William, Lusk Ewing and Skjellum Anthony. *Using MPI: Portable Parallel Programming with the Message Passing Interface Second Edition.* MIT Press, 1999.

[Haw97] Hawick K. A. Trends in High Performance Computing. In *Proceedings of 4^{th} IDEA Workshop, Magnetic Island*, 17-20 May 1997.

[Hic98] Hicks Michael, Kakkar Pankaj, Moore Jonathan T., Gunter Carl A. and Nettles Scott. PLAN: A Packet Language for Active Networks. In *Proceedings of the 3^{rd} ACM SIGPLAN International Conference on Functional Programming language*, pages 86–93, ACM, 1998.

[HMA+99] M. Hicks, J. Moore, D. Alexander, C. Gunter, and S. Nettles. Planet: An active internetwork, 1999.

[Kat97] Katz D. IP Router Alert Option. *RFC 2113, IETF*, February 1997.

[Ler00] Leroy Xavier. A Modular Module System. *Journal of Functional Programming*, pages 10(3):269–303, 2000.

[Lit87] Litzkow Michael J. Remote Unix: Turning Idle Workstations into Cycle Servers. In *Proceedings of the USENIX Summer Conference. USENIX*, June 1987.

[Lit88] Litzkow M., Livny M. and Mutka M.W. Condor – A Hunter of Idle Workstations. In *Proceedings of 8^{th} International Conference on Distributed Computing Systems*, pages 108–111, june 1988.

[Mut87] Mutka M. W. and Livny M. Scheduling Remote Processing Capacity in a Workstation-Processor Bank Network. In *Proceedings of 7^{th} International Conference Distributed Computing Systems, Berlin, West Germany*, pages 2–9, 1987.

[Nyg99] Nygren Erik L., Garland Stephen J. and Kaashoek M. Frans. PAN: A High-Performance Active Network Node Supporting Multiple Mobile Code Syatems. In *Proceedings of OpenArch'99, NewYork*, pages 78–89, March 1999.

[Ode97] Odersky Martin and Wadler Philip. Pizza into Java: Translating Theory into Practice. In *Proceedings of 24^{th} ACM Symposium on Principles of Programming Languages, Paris, France*, January 1997.

[Pac98] Pacheco P. S. A User's Guide to MPI. Technical report, Department of Mathematics, University of San Francisco, 1998.

[Pfi98] Pfister G. *In Search of Clusters, 2^{nd} Edition*. Prentice Hall, 1998.

[Pos81] Postel J. Inetrnet Protocol. *RFC 792, ISI*, September 1981.

[Raj99a] Buyaa Rajkummar. *High Performance Cluster Computing (Vol. 1): Architectures and Systems*. Prentice Hall, 1999.

[Raj99b] Buyaa Rajkummar. *High Performance Cluster Computing (Vol. 2): Programming and Applications*. Prentice Hall, 1999.

[Res96] Reschke C., Sterling T., Ridge D., Savarse D., Becker D. and Merkey P. A Design Study of Alternative Network Topologies for the Beowulf Parallel Workstations. In *Proceedings of the 5^{th} IEEE International Symposium on High Performance Distributed Computing*, 1996.

[Rid97] Ridge Daniel, Becker Donald, Merkey Phillip and Sterling Thomas. Beowulf: Harnessing the Power of Parallelism in a Pile-of-PCs. In *Proceedings of IEEE Aerospace Conference*, 1997.

[Riv92] Rivest R. The MD5 Message-Digest Algorithm. *Request for Comments: 1321*, April 1992.

[Sch99] Schwartz B., Jackson A., Strayer T., Zhou W., Rockwell R. and Partridge C. Smart Packets for Active Networks. In *Proceedings of OPENARCH'99*, March 1999.

[Sta96] Stallings William. *SNMP, SNMPv2 and RMON: Practical Network Management*. Addison-Wesley, 2^{nd} edition, 1996.

[Sun87] Sun Michrosystems, Inc. XDR: External Data Representation Standard. *RFC 1014, Sun Michrosystems, Inc.*, June 1987.

[Ten96] Tennenhouse David L. and Wetherall David J. Towards an Active Network Architecture. *Computer Communication Review, 26(2)*, 1996.

[Ten97] Tennenhouse D. L., Smith J. M., Sincoskie W. D., Wetherall D. J. and Minden G. J. A Survey of Active Network Research. In *Proceedings of IEEE Communications Magazine*, pages 80–86, january 1997.

[Wal92] Walsch A., Schulz M. W., Lindenstruth V. SCI: Scalable Coherent Interface. In *IEEE Standard 1596-1992*, 1992.

[Wet96] Wetherall David J. and Tennnenhouse David L. The ACTIVE IP Option. In *Proceedings of the 7^{th} ACM SIGOPS European Workshop, ACM, Connemara, Ireland*, September 1996.

[Wet97] Wetherall D. Developing Network Protocols with the ANTS Toolkit. *ANTS Distribution, Design Review*, 1997.

[Wet98] Wetherall D., Guttag John, and Tennenhouse David. ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols. In *Proceedings of 1^{st} Open Architectures and Network Programming Conference (OPENARCH'98),San Francisco, CA*, pages 117–129, April 1998.

[Wet99a] Wetherall David. Active Network Vision and Reality: Lessons from a Capsule Based System. In *Proceedings of 17^{th} ACM Symposium on Operating System Principles (SOSP99), pages 64-79*, 1999.

[Wet99b] Wetherall David J. *Service Introduction in an Active Networks*. PhD thesis, Massachusetts Institute of Technology, February 1999.

[Wol97] Wolski Rich. Forecasting Network Performance to Support Dynamic Scheduling Using the Network Weather Service. In *Proceedings of 6^{th} Intl. Symposium on High Performance Distributed Computing (HPDC-6)*, 1997.

[Wol98] Wolski Rich. Dynamically Forecasting Network Performance Using Network Weather Service. *Journal of Computing, Volume 1*, pages 119–132, January 1998.

[Wol99] Wolski Rich, Spring Neil and Hayes Jim. The Network Weather Service: A Distributed Performance Forecasting Service for Metacomputing. *Journal of Future Generation Computing Systems, Volume 15, Numbers 5-6*, pages 757–768, October 1999.

[Yem91] Yemini Y., Goldszmidt G. and Yemini S. Network Management by Delegation. In *Proceedings of the 2nd International Symposium on International Network Management, Washington DC*, 1991.

[Yem96] Yemini Y. and da Silva S. Towards Programmable Networks. In *Proceedings of Workshop on Distributed Systems: Operations and Management, L'Aquila, Italy*, 1996.