UNIVERSITY OF CALIFORNIA, SAN DIEGO

Large Scale Autonomous Computing Systems

A dissertation submitted in partial satisfaction of the requirements for the degree Doctor of Philosophy in Computer Science

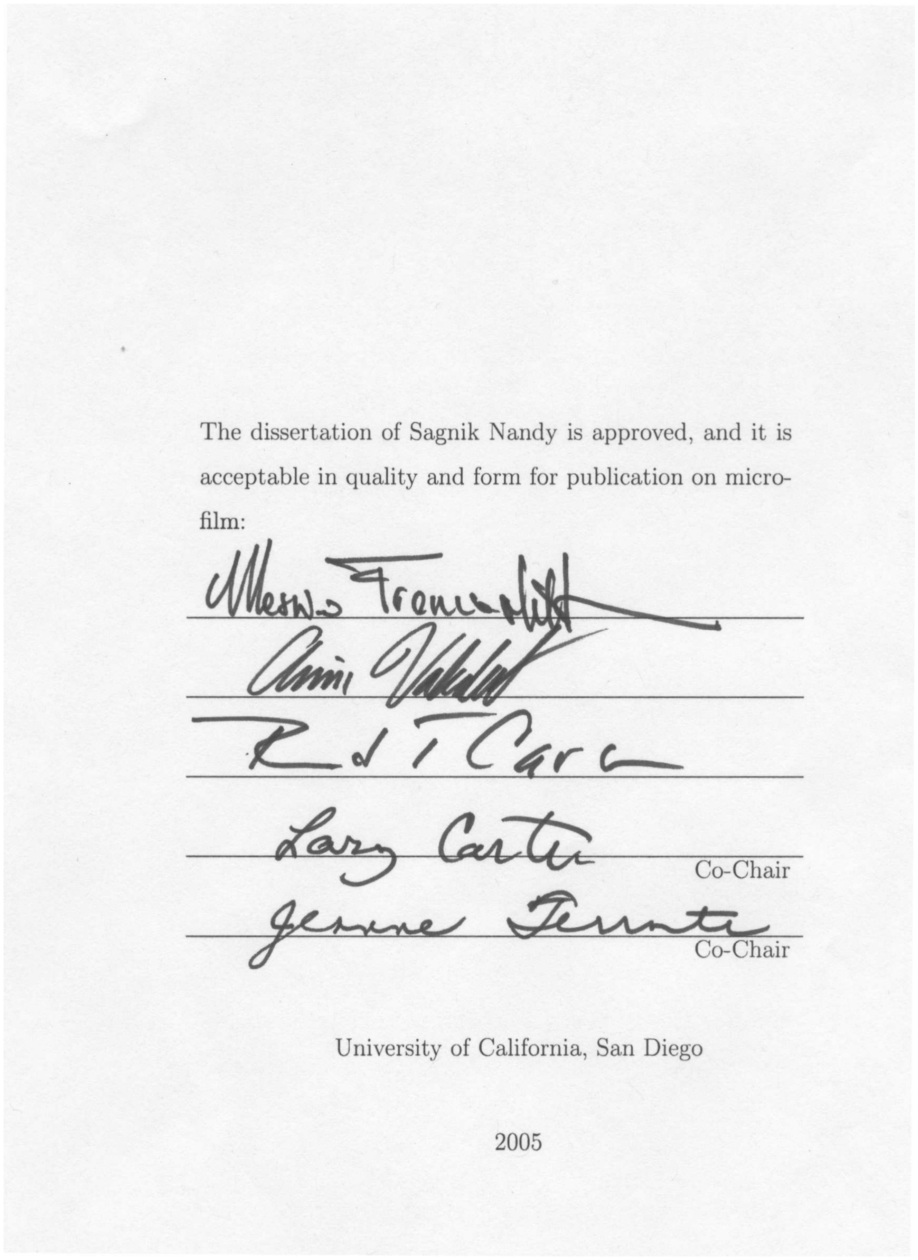
by

Sagnik Nandy

Committee in charge:

Professor Larry Carter, Co-Chair
Professor Jeanne Ferrante, Co-Chair
Professor Amin Vahdat
Professor Richard Carson
Professor Massimo Franceschetti

2005



The dissertation of Sagnik Nandy is approved, and it is acceptable in quality and form for publication on microfilm:

[illegible]

[illegible]

[illegible]

[illegible] Co-Chair

[illegible] Co-Chair

University of California, San Diego

2005

DEDICATION

This dissertation is dedicated to all the people who have influenced me and helped me be the person that I am today.

“You are never given a dream without also being given the power to make it true.”

– Richard Bach

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to thank the following people and institutions for the immense support and encouragement they provided. Without them I wouldn't have been the person that I am today and this thesis wouldn't have been a reality.

My family - my mother (Mummum), father (Bubin), sister (Pummy) and my grandparents, for everything that they have done for me and for the immense belief they have shown in me. I would specially like to thank my mother, who has been the greatest influence on my life. She was the one who taught me to read, write and think and without her love, blessings and guidance I wouldn't have been the person that I am today.

St. James's School, Calcutta, where I spent the most amazing and formative fifteen years of my high school life. I would like to thank all my friends from high school for the great time I had there as a student. I have been lucky to have had some of the best teachers one could possibly ask for. My teachers in high school were an immense source of inspiration, encouragement and support. They treated us like one big family and I feel fortunate to have been a part of that family. I would specially like to thank Mr. Abhijit Sirkar, my computer teacher, for introducing me to the area of Computer Science and making me fall in love with it.

The faculty and students in BITS (Birla Institute of Technology and Science), Pilani, where I did my undergraduate studies. It was here that my love for Computers took greater shape. If St. James's School gave me the ability to think clearly, it was BITS that gave direction to those thoughts. BITS also made me a more responsible person and introduced me to some really talented people who inspired me to achieve.

My friends Nileen, Abhik, Ranajit, Haimanti, Megha and Arijit for always being there for me. I would specially like to thank Nileen, who is one of the most talented people I have ever come across. His talents and humility have been a tremendous source of inspiration.

The faculty and students of UCSD for making graduate student life so much fun. The past few years have really taught me a lot and have also made me appreciate and enjoy Computer Science a lot more. I am really thankful for the wonderful experience leading to this thesis and it has been an absolute honor to be a part of UCSD.

My friends Sid, Vipul, Siddhartha, Sudipta, Subhradyuti and Rishi for making my stay in San Diego so easy and enjoyable.

Everyone in the HPCL group at UCSD, especially Shelly, Sean, Barbara, Mike and Xiaofeng.

Professor Amin Vahdat, Professor Richard Carson and Professor Massimo Franceschetti for agreeing to be in my defense committee and for the valuable suggestions and feedback that they provided.

Finally, my fantastic professors, Professor Larry Carter and Professor Jeanne Ferrante. I had read the following lines about Larry in an earlier thesis - "I not only worked for a great scientist, but a great person" and I couldn't have said it better. He has had one of the greatest influences on me. I have learnt a tremendous amount from him just by observing and interacting with him. Larry taught me that every problem, irrespective of the field they are associated with, is interesting and worth solving. This has been one of the most valuable lessons that I have learnt as a student. Jeanne has had an equally important influence on me. She is one of the warmest human beings that I have come across and has always helped me through every problem that I faced here in UCSD. Jeanne always made me feel like a friend first and then a student and I am really thankful to her for that.

This work was supported by the CAL-IT(2) fellowship and in part by the NSF grant ACI-0234233.

Portions of the text of Chapters 3 and 4 are a reprint of the material as it appears in [72, 71]. The dissertation author was the primary researcher and author and the co-authors listed on these publications directed and supervised the

research which forms the basis of these chapters.

VITA

1978	Born, Calcutta, India
1997	High School Diploma, St. James' School, Calcutta, India
1997–2001	B.E. (Hons) Computer Science and Engineering, BITS (Birla Institue of Technology and Science), Pilani, India
2001–2002	Cal-IT(2) Fellowship
2002–Present	Research Assistant, University of California, San Diego
2003	M.S., University of California, San Diego
2003	Summer Intern, IBM T.J. Watson Research Center, New York
2005	Teaching Assistant, University of California, San Diego
2005	Doctor of Philosophy, University of California, San Diego

PUBLICATIONS

"TFP: Time-Sensitive, Flow-Specific Profiling at Runtime." S. Nandy, X. Gao, and J. Ferrante, In the 16th Workshop on Languages and Compilers for Parallel Computing (LCPC), October, 2003.

"A-FAST: Autonomous Flow Approach to Scheduling Tasks." S. Nandy, L. Carter, and J. Ferrante, In the Proceedings of the 11th Interantional Conference of High Performance Computing (HiPC), December, 2004.

"GUARD: Gossip Used for Autonomous Resource Detection." S. Nandy, L. Carter, and J. Ferrante, In the Proceedings of the 19th International Parallel and Distributed Processing Symposium (IPDPS), April, 2005.

"Interference-Aware Scheduling." B. Kreaseck, L. Carter, H. Casanova, J. Ferrante, and S. Nandy, In the International Journal of High Performance Computing Applications (IJHPC), Vol 20, February, 2006.

FIELDS OF STUDY

Major Field: Computer Science

Studies in Distributed Parallel Computing.
Professors Larry Carter and Jeanne Ferrante

Studies in High-Performance and Scientific Computing.
Professor Larry Carter

Studies in Compiler Technology.
Professor Jeanne Ferrante

ABSTRACT OF THE DISSERTATION

Large Scale Autonomous Computing Systems

by

Sagnik Nandy
Doctor of Philosophy in Computer Science
University of California, San Diego, 2005
Professor Larry Carter, Co-Chair
Professor Jeanne Ferrante, Co-Chair

The growth in size and popularity of the Internet has led to several wide area computing efforts where the computing power of several thousands of machines are combined together. As these systems grow in size, it becomes increasingly difficult to control and monitor them using a client-server-like architecture. Introducing decentralization and autonomy in these systems seems to be the obvious solution. Autonomy has several advantages. These include easy scaling, removal of single points of failure, cost-effective deployment and greater promotes collaborative environments. However, autonomous systems present certain challenges as well. The absence of server-like nodes make it difficult to locate and manage available resources, schedule tasks efficiently across them, assure security and develop applications that can make use of these powerful platforms. This dissertation addresses some of these issues. We present autonomous techniques for resource location, task scheduling and programmability in large scale computing systems. Our work tries to answer the question "*Do autonomous solutions of these issues achieve same (or comparable) performance benefits as their centralized counterparts?*".

We propose GUARD, a protocol for locating computing resources in a completely autonomous environment. GUARD can locate resources matching single or multiple parameters and it provides a trade off between efficiency and probability of success.

We then discuss A-FAST, a decentralized scheduling technique for homogeneous independent tasks, that uses the notion of pressure from fluid networks to autonomously guide tasks to resources. Simulations show A-FAST to achieve a high level of efficiency ($\approx 98\%$ of optimal steady state throughput) for both computation and communication dominated tasks.

The dissertation also discusses a programming platform that we designed and implemented to allow the development of applications for decentralized environments. Our platform allows programmers to develop code without having knowledge of the underlying system and allows platform-specific information to be fed to the program at runtime. We show how this platform can be used to generate code that adapts to several situations without changing the original program or involving the programmer in the scheduling decisions.

Chapter 1

Introduction

The past few decades have witnessed a great deal of research towards the development, maintenance and deployment of large scale distributed computing systems. These systems have grown from clusters of local machines [95] to collections of such clusters [64, 33] to hundreds of thousands of isolated machines connected over the Internet [91, 29]. This has made it possible to harness the strength of thousands of isolated machines to produce computing power that is comparable to the fastest of supercomputers [1].

As these systems grow in size and cross administrative boundaries, it becomes increasingly difficult to control and monitor them using a strictly client-server like architecture where a single (or group of) powerful machines control the functioning of the entire system. Client-server systems, where a set of powerful machines control the functioning of the other machines in the system, are often expensive to build and maintain (due to the expensive hardware and software needed for the servers) and the server-like nodes are often the bottleneck of the systems. One can therefore witness a growing trend of introducing decentralization in these systems, shifting them from strictly centralized systems [91, 29, 89] to hierarchical [40, 3] and even completely decentralized systems [98, 35, 41, 74, 77]. Decen-

[1] e.g. In 2002, SETI@home could operate at 50 TFlops, while IBM Blue Horizon operated at about 36 TFlops.

tralization, where the control of the system is spread across all the nodes of the system, helps in removing single points of failure from the system, eases scalability, reduces the cost of deployment and maintenance and allows better collaborative efforts (by allowing isolated users to set up collaborative environments without requiring expensive hardware and software). However, decentralization introduces several additional challenges. The absence of a centralized monitoring mechanism makes it difficult to locate resources (since there is no centralized repository with knowledge of all the individual nodes and their abilities), assign tasks across them (since a scheduler might not be even aware of the overall strength and ability of the underlying system and how they will change with time), ensure security (due to the absence of a centralized authentication mechanism) and provide adequate support for programmability (the programmer now has to develop code for a possibly unknown and dynamic system). In this dissertation we try to address some of these challenges by developing autonomous (defined in Chapter 2) solutions for them. Specifically, we address the following three questions with respect to decentralization and large scale computing systems:

- **Resource Allocation**: How to keep track of and locate computing resource(s) meeting a given criteria, in the absence of centralized or hierarchical book keeping? We present GUARD (Gossip Used for Autonomous Resource Detection), a protocol that locates computing resources in a completely autonomous fashion.

- **Resource Scheduling**: How to schedule a pool of homogeneous equal-sized tasks across a heterogeneous and dynamic large scale system so that the steady state throughput is maximized? We present A-FAST (Autonomous Flow Approach to Scheduling Tasks), a protocol that uses the notion of pressure in fluid networks to schedule tasks in a completely decentralized manner.

- **Programmability**: What functionalities does a programmer need in order to develop code for a large scale distributed system that is changing and doesn't have centralized nodes monitoring it? How can one hide the system specific issues from the programmer and allow development of code that is independent of the underlying system? We discuss the design and implementation of a platform that provides programmers with a simplified view of the system, thereby hiding several underlying system complexities associated with decentralization.

The dissertation describes decentralized and autonomous techniques that address the above three issues while trying to maintain the advantages and efficiency of a centralized solution. Our work supports the claim that:

Autonomous resource locating, task scheduling and programmability achieve the same (or comparable) performance benefits as their centralized counterparts while achieving the benefits of decentralization.

For resource locating, GUARD (Gossip Using Autonomous Resource Detection), uses gossiping and distance vectors to locate computing resources in a decentralized scenario. GUARD can find resources meeting a single or multiple criteria and can achieve a desired level of success (finding a resource using the least number of hops) by varying the gossiping frequency alone (Section 3.2.2).

For task scheduling we discuss A-FAST (Autonomous Flow Approach to Scheduling Tasks), which functions inspired by how pressure controls flows in fluid networks. A-FAST schedules a pool of homogenous tasks across a heterogeneous, dynamic and large scale system and works well (achieves $\approx 98\%$ of optimal steady state throughput) for both computation and communication based systems. A-FAST also captures system issues like reliability. We implemented A-FAST in a real system and present experimental results.

For programmability we designed and built a system architecture that clearly separates the roles of the scheduler, the message handler and the program.

These three components then interact using a simple but strictly defined interface that can feed runtime information to the program. The programmer can thus be unaware of the complexities of the underlying system and the scheduler can take the necessary actions. Experiments show that our architecture deals with several different underlying issues without involving the programmer in the decision making. This allows the programmer to write code only once and run it efficiently across a range of platforms.

We hope that this work will help towards building large-scale decentralized computing systems. Decentralized computing has made it possible for us to build inexpensive but highly powerful computing systems that are solving a large range of important problems. We hope that our work will complement existing and ongoing research activities that seek to harness the true potential of the World Wide Web, by allowing to people to connect their computers over the Internet and build huge computing communities.

The rest of this thesis is organized as follows. Chapter 2 defines Autonomous Computing Systems and describes their advantages, disadvantages, and the various issues involved in using them. Chapters 3 and 4 describe GUARD and A-FAST in detail. In Chapter 5 we discuss the issue of programmability in distributed systems and provide our results associated with it. Chapter 6 takes a look at the related research efforts in this area. Finally we end with a summary, a discussion of the various lessons learned, and possible future work in this area in Chapter 7.

Chapter 2

Autonomous Computing Systems

This dissertation aims at providing autonomous and decentralized solutions to some issues concerning distributed computing systems. It is therefore important to clearly define what an autonomous or decentralized computing systems is and describe their advantages and challenges; this is the aim of the current Chapter. We also look at the various degrees of autonomy that some of the present distributed computing systems exhibit by presenting some case studies.

2.1 Definition of Autonomous Computing Systems (ACS)

A "decentralized computer system" is defined in [43] as "a collection of autonomous computers which communicate with one another to perform a common service". By *Autonomy* we mean the ability to function independently without control by other entities. Thus all the members of a Autonomous Computing System (ACS) should ideally be able to *control* and *monitor* their own activity, with no (or minimal) external intervention.

In our view autonomy is directly connected to decentralization and the degree of autonomy a system exhibits is a direct measure of how decentralized it is. For the rest of this dissertation we will use the terms decentralization and autonomy interchangeably.

A growing group of modern systems that possess a high degree of autonomy are the peer-to-peer content sharing systems [56, 73, 35]. Peer-to-peer computing is defined as the sharing of computer resources and services by direct exchange between systems. There are several similarities between "peer-to-peer computing" and "autonomous computing", the terms are often used interchangeably. However, even though these paradigms share several similarities, in our view they are not the same. Peer-to-peer systems allow participants to directly interact with each other but do not necessarily remove centralized intervention. For example, Napster [73] and Javelin [74] support peer-to-peer paradigms while retaining some form of centralized control. For the context of this dissertation, we would judge the "degree of autonomy" present in a system based on the amount and kind of decentralized functioning it exhibits in the following categories:

- **Resource Discovery**: To benefit from the vast range of resources that distributed systems have to offer (CPU-cycles, memory, storage, software etc.), it is necessary to discover and access these resources. Autonomous resource discovery implies that nodes in the system should not make use of centralized server(s) or global knowledge to locate the system's available resources, but instead rely on local information to accomplish what's needed.

- **Resource Control**: One of the main reasons for having centralization in distributed systems is to control and monitor the functioning of resources available to the system. Centralized server(s) or server-like nodes determine what the functionality of the participating nodes in the system should be and pass on the corresponding control instructions. In a fully autonomous environment, the task of resource control is spread across all the nodes of the system, leading to an egalitarian computing environment.

- **Resource Interaction**: For the functioning of any distributed system, individual components of the system need to interact with each other and exchange information. Traditional systems based on the client-server paradigm often have the server mediating the communication and interaction between system components. This can hinder their autonomy. For a system to be truly autonomous, components should be able to interact directly with each other without any central mediation.

To summarize, *an Autonomous Computing System (ACS) is a collection of computational nodes that are capable of functioning independently without the presence of centralized server nodes. Nodes in such a system are only aware of their immediate neighbors and do not have a global view of the system. All decisions are made based on this local information, and interaction requires no centralized intervention.*

Having described what an Autonomous Computing System is, we now take a look at some of the principal advantages of using such a system.

2.2 Advantages of an ACS

The growing trend of autonomy in computing systems [55, 73, 98] supports the claim that autonomy is an attractive proposition. In this section we outline advantages associated with building autonomous computing systems. The include: scalability (2.2.1), fault tolerance (2.2.2), economics (2.2.3) and community building (2.2.4).

2.2.1 Scalability

To make use of the growing number of computing resources around us, a system has to be scalable. Scalability can be defined as the ability of a system to handle an increase or decrease in resource volume without significant performance degradation per node. For years researchers have worked towards improving scalability of distributed systems. A single server managing these systems can only support a limited number of users, limiting the system's scalability scalability. Thus as the number of computing nodes increase in the system, the number of nodes that interact with the server also increases, thereby affecting the overall performance. With decentralization, the task of managing and monitoring the system gets delegated amongst all the participants, thereby reducing (or eliminating) the role of centralized servers. This allows the systems to scale to greater sizes.

2.2.2 Fault Tolerance

As described in the previous segment, in a true ACS, each component exhibits autonomy and is capable of making its own decisions. The functional interdependence between components in these systems is therefore minimized and any component can be inserted/removed from the system without affecting the functioning of other components. Therefore, in an ideal ACS, any subset of the system should also be able to function independently as a complete system. This reduces the chances of a single member (or group of members) bringing down the system completely, improving the fault resilience of the overall system. Distributing the components of the system also help in eliminating single points of failure due to malicious attacks. This helps in reducing the vulnerability of the system, making it more reliable.

2.2.3 Economics

One of the biggest advantages of building ACSs is the reduction in the cost of set up and deployment. Traditionally servers (or server-like nodes) require powerful hardware and software support along with regular maintenance. This often makes these components a lot more expensive than the participating computing nodes[1]. By delegating the server's responsibilities among several low-end machines, the cost of maintenance, along with the need for expensive hardware and software to perform the specialized tasks, can be reduced considerably.

2.2.4 Greater Community Building

One drawback of many existing distributed computing systems is that they do not support the growth of online communities. This is primarily due to the expensive hardware, software and technical expertise often associated with building these communities, restricting participation of users who are willing to collaborate and build their own communities, but are not part of organizations supporting such efforts. With the development of ACSs, the tasks performed by high-end systems would get managed locally by the participating machines, facilitating loosely knit isolated users to build their own computing communities seamlessly. The popularity of some of the peer-to-peer computing systems [55, 12] have supported this claim by allowing millions of isolated users to seemlessly form their own communities over the Internet.

2.3 Challenges

Given the advantages of Autonomous Computing Systems, the obvious question to ask is - why haven't they been implemented widely till now? Even though ACS as a concept is appealing, there exist several challenges in implementing them in practice. This section presents some of the basic requirements of distributed computing systems and identifies the challenges associated with meeting these requirements in a decentralized fashion.

[1]A single high end 8-way server costs around $750,000 whereas a rack comprising of 100 PCs costs only around $300,000.

2.3.1 Resource Discovery

To be part of a ACS and use its resources, one has to first discover these resources. In a computing environment, some of the resources one wishes to discover may even be dynamic in nature (e.g. a machine with less than two processes running). Several existing systems have used centralized registration and monitoring mechanisms to keep track of resources and their current states [64, 34]. Resources can then be located in these systems by querying the centralized server(s). In a decentralized environment, there are no global repositories maintaining system-wide resource information and status, making resource discovery a more challenging issue.

2.3.2 Security

One of the biggest issues in distributed computing is security. By making a system available for computation, one opens it to a range of vulnerabilities. Code executing on a participating systems can cause mayhem (damage the system, leak system data etc.). It is therefore the responsibility of the system to ensure that the participants are protected from malicious users and code. A common way of dealing with security in distributed systems is to have trusted member(s) from whom users download code. These members can either be well established names that users trust [91, 29], or can take it upon themselves to ensure the safety of the executing code [40]. In a truly decentralized environment, there exists no globally trusted and secure member, putting the responsibility of security entirely upon the members themselves.

2.3.3 Correctness

Correctness refers to the integrity of the results of a computation. Some research efforts associate correctness with security. However, security deals with the danger caused to the system by the executing code rather than the incorrect execution of code. Popular techniques like Polling and Spot-checking can be implemented in a decentralized manner to ensure correctness of results. However, these techniques also try to identify the sources of the anomalies in order to blacklist them. In the absence of centralized control, tasks may travel across the system autonomously, making it more difficult to identify the actual source(s) of the errors.

2.3.4 Scheduling

Several distributed systems target performance by simply providing additional resources. However, since resources are expensive, one would ideally like to make the best use of these resources. Efficient scheduling leads to better utilization of resources. Several systems make use of centralized schedulers, which maintain system-wide information (e.g. processor speeds, network topology etc.) to assign tasks to workers [64, 40, 23]. Though this leads to better resource utilization, the centralized controllers often become the bottlenecks for scalability. ACS solves the problem of scalability but makes it more challenging to manage resources efficiently without global knowledge and external supervision.

2.3.5 Programmability

One factor which is likely to determine the future and popularity of distributed systems is the amount of programmability it offers to the user. Programmability refers to the kind of programming support the system offers and the kind of applications that can be run on it. A large range of distributed applications require some form of centralized resource (e.g. shared memory) or involve some variant of centralized control (e.g. BSP [17], divide- and-conquer etc). An ACS has no knowledge of global resources, making it difficult to support these programming models. Moreover, many of the existing programming models [70] assumes the programmer has an idea of the size of the system, and are built to perform well only with homogeneous resources. A large scale ACS is likely to be extremely dynamic in nature and the components of the system are likely to heterogeneous in terms of ability and availability. This makes the task of the programmer a lot more difficult, since several system specific complexities then have to be taken care of.

2.3.6 Fault Tolerance

Recent studies [40, 16] have shown node failures in distributed systems to be commonplace. This, coupled with the long running times of several distributed applications, makes fault tolerance a critical issue. Like programmability, many existing systems use centralized control to detect and deal with failures [40] or use centralized resources (e.g. shared tuple space) to reduce the effects of these failures [20]. Decentralization makes the overall system more fault tolerant, by removing single points of failures, but makes application fault tolerance more challenging, since applications can no longer rely on centralized controllers to detect and deal with failures.

2.4 Degrees of Autonomy

A distributed system need not be completely autonomous. Nor does it have to be completely centralized. One can imagine these to be two ends of a spectrum. Most existing distributed systems exhibit some form of autonomy in the categories mentioned in Section 2.1, with the degree of autonomy varying. To understand the contribution ACS's further and appreciate the complexities involved in designing solutions for them it is important to study some existing computing systems and see how they fare in terms of the autonomy they exhibit. We present four computing systems - SETI@home, XTremWeb, P^3 and the Javelin system, each with varying degree and forms of autonomy. Of these, SETI@home is almost a strictly client-server system, while Javelin is closest to our definition of an ACS.

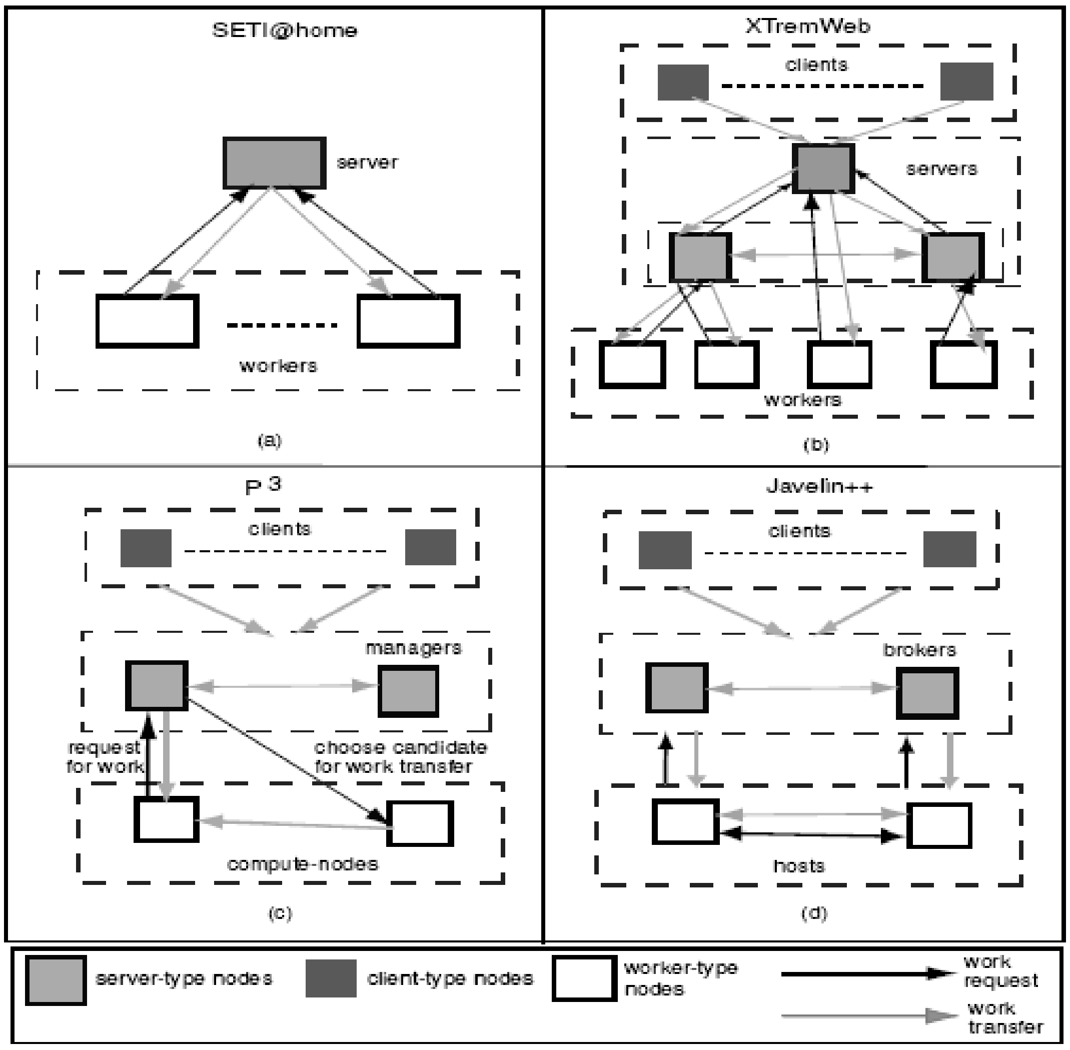


Figure 2.1: High level architecture of the four examples (a) SETI@home - interactions are strictly client-server; (b) XTremWeb - client-server model with servers arranged hierarchically; servers in same layer interact in a peer-to-peer fashion; (c) P^3 - managers exchange tasks among themselves in a decentralized fashion; a manager can (i) send a task to the requesting compute-node directly or (ii) locate another compute-node to share its task, in which case the task is transferred directly between the two compute nodes; and (d) Javelin - brokers can share tasks in a decentralized fashion; hosts ask brokers as well as other hosts for work; thus unlike the other systems, hosts can directly ask each other for work.

2.4.1 SETI@home

Overview: SETI (Search for Extraterrestrial Intelligence) [4, 91] is a project that harnesses the power of idle desktop machines in order to search for alien life forms. It is one of the biggest examples of public resource computing (it has a user base of nearly 4.5 million in 2002). The system is built upon a client-server architecture as shown in Figure 2.1. SETI's simple architecture and novelty of purpose have been the main reasons for its enormous popularity. Other projects with similar structure include distributed.net [29] and the GIMPS [68] project.

Analysis: SETI achieves resource discovery through the centralized server. All users log on to this server, and it is responsible for controlling and monitoring all the clients. Interaction in the system is strictly between the client and the server as well. Thus the system does not meet any of the requirements of decentralization stated in Section 2.1. Yet, SETI is one of the most successful and popular distributed computing projects. In addition, the client-server architecture of SETI helps it solve problems of security (trusted server is the only source of code) and correctness (the server alone is responsible for correctness and achieves it by redundancy). This raises the question of whether there is any need for decentralizing distributed computing systems. To answer this question we evaluate SETI with the benefits discussed in Section 2.2.

Scalability has not been a major issue for SETI since there are no strict deadlines and the server can make a client wait indefinitely before servicing it. Moreover, the computation to communication ratio of the system is very high, and it doesn't matter how fast the jobs are scheduled, reducing the responsibility of the server further. However, the single server architecture has started causing serious problems in outgoing bandwidth, with performance drops as high as 25% for the system. Decentralizing the system would allow users to download tasks from each other - reducing the bandwidth congestion of the server.

Having a single centralized architecture also makes SETI more sensitive to faults since any damage to the server can cause the system to halt. However, due to the loose deadline on the computation, the overall performance of the system is not affected significantly. Nevertheless, decentralizing the system would increase its resilience against faults.

Coming to the issue of economics - SETI is definitely not an easy project to launch and manage. Nearly the entire cost of the project is due to the server and back-end data base. Decentralizing it would definitely make the system more economical to build and manage by allowing client machines to share some of the responsibilities of the server.

Moreover, due to its single application approach, SETI does not provide any support for programmability. The collaborative environment is one-sided, i.e. users can only contribute resources but not jobs themselves. Thus even though SETI is a successful system, it is monolithic in structure and does not support several desired features of distributed computing systems. BOINC [3], a very recent follow up of SETI, tries to overcome most of these shortcomings and has already identified decentralization and autonomy as key issues. To summarize, SETI@home is an example of an extremely successful system that does not use the notion of decentralization. The centralized architecture and dedicated server solves several challenges for the system, but does so at the expense of flexibility and generality. General purpose applications, therefore, are unlikely to benefit much from a SETI-like platform.

2.4.2 XTRemWeb

Overview: The XTremWeb system [40, 39] is a global computing effort that uses a user-server-worker model. Unlike SETI, users in XTremWeb can submit tasks themselves. These tasks are initially sent to XTremWeb servers, which use dispatchers to transfer them to schedulers. Schedulers are responsible for assigning the tasks to collaborating workers, which execute them. Scalability is addressed by having multiple servers arranged in a hierarchical fashion. Issues such as security, interoperability and fault tolerance are also addressed by the servers. For security, servers (or dedicated machines) test the code on themselves before sending it to the workers. Interoperability is achieved by maintaining multiple binaries of the jobs at the server and assigning them to compatible host machines. For fault tolerance the servers monitor the "liveness" of workers, re-sending work if needed.

Analysis: XTremWeb, like SETI, follows a client-server like model. However, unlike SETI, it has a hierarchical system structure, thereby making it more scalable and robust.

XTremWeb achieves resource discovery in a centralized fashion using servers. The servers and schedulers exercise complete control on the functioning of the workers, monitoring them and determining which tasks they execute. Therefore, using our definition of decentralized systems, XTremWeb is predominantly a centralized system. However, resource interaction in XTremWeb exhibits some element of decentralization, since servers in the same layer are allowed to communicate directly with each other. Thus within the server peer group the interactions are decentralized, and the advantages of this are evident - the system achieves better load balancing (by distributing work to other servers) and fault tolerance (by replicating work to other servers) without creating a central bottleneck. However, the user-worker interaction is still centralized, controlled by the servers. This makes the XTremWeb server-layers the bottleneck of the system.

The hierarchical and semi-centralized architecture of XTremWeb has its advantages - it achieves better scalability than completely centralized systems and still provide a high degree of security and fault tolerance (by having dedicated servers for this purpose). The main drawback, however, is the cost associated with building the system. The XTremWeb servers are dedicated and persistent machines supported by back-end data base support. With growing number of users, the need for these servers will increase. This will affect the cost of deploying large-scale XTremWeb systems. Decentralizing the system will reduce the need for servers by distributing part of their work to the workers, bringing down the cost of the system.

To summarize, XTremWeb is a good hybrid solution for connecting clusters of machines but is not well suited for scenarios where one wishes to combine large number of isolated machines for distributed computing.

2.4.3 P^3 : Parallel Peer-to-peer

Overview: The aim of P^3 [77], was building an ACS. The system has a layered structure comprising of clients, managers and compute- nodes. The manager nodes are the more "capable" nodes with greater resources (in terms of network connection, storage, processing speed etc.) and are responsible for controlling the functioning of the system, while the compute nodes are less resourceful and provide temporary storage and processing cycles. Jobs are initially submitted by the clients to the manager layer. Compute nodes request work from managers. The managers either assign the node a new job or direct another compute node to share its work with the requesting node. In the latter situation all communication between the two compute nodes take place directly, in a peer-to-peer fashion. The manager layer just identifies the source and target of a computation but does not involve itself in the actual task transfer.

P^3 also provides virtual shared memory (Object Space) support using a decentralized distributed file system. This provides several benefits such as (i) synchronization primitives to provide better programmability (ii) meta-data service for the managers to achieve better scheduling and (iii) checkpointing support for fault tolerance.

Analysis: P^3 strongly supports the idea of decentralization. The Object Space allows decentralized storage and retrieval of data in the system. This helps in discovering resources such as code, checkpoint information and metadata. in a decentralized fashion. Task transfer between compute nodes take place in a decentralized manner as well, without going through a manager. P^3 thus exhibits decentralization in resource discovery and interaction, qualifying as a largely decentralized system by our definition. However, the system still has a somewhat centralized approach towards resource control. Even though task transfer between compute nodes take place in a decentralized fashion, all transfers are initiated by the manager layer. Managers thus exercise complete control over where a task executes and when to convert a compute node to a manager node. This prevents the compute nodes from exercising functional autonomy. Thus though P^3 is predominantly a decentralized system it still has some elements of centralized control in it.

Coming to the advantages of decentralization in P^3, the ability to directly transfer tasks between compute nodes prevents managers from becoming communication bottlenecks. For fault tolerance, manager nodes use storage in other nodes as well to save check-pointed states. This reduces the required capability, responsibility and resources needed by manager nodes, allowing them to create additional managers dynamically from compute-nodes, making the system more scalable. The need for expensive and powerful machines is also reduced. In a system such as XTremWeb, the servers are completely responsible for security, communication, scheduling, fault tolerance, portability etc. Such servers often must be high-end machines, preventing ordinary host machines to substitute for them. P^3 delegates part of the server's work to the compute nodes (fault tolerance, communication) leading to more scalable and cost effective systems.

To summarize, P^3 has a layered design but introduces a fair degree of decentralization within the layers. The system also reduces the differences between its layers, thereby reducing the centralized control between them.

2.4.4 Javelin

Overview: The Javelin system [75, 74] is an initiative to support Java-based large-scale parallel computing for adaptively parallel applications. The system is based on a client(task provider)-broker- host(resource provider) architecture as shown in Figure 2.1. The broker layer, like the manager layer in P3, provides centralized control and monitors issues such as scheduling and resource discovery. In the initial version of Javelin, client applications were written as Java-applets and uploaded in a server. This solved the problems of portability (by making use of the JVM) and security (by trusting Java's built-in security mechanism for applets). A Cycle Stealing approach, as introduced in the Cilk [13] system, was used for scheduling, while fault tolerance was achieved using Eager Scheduling [10].

Javelin++, a follow up to the initial version, added further features to the system. These included support for Java applications, a distributed broker network to add scalability, and two different scheduling approaches - deterministic, where hosts are arranged in a tree structure, and probabilistic, where each host maintains a list of some other hosts and can perform cycle stealing directly without intervention from the broker. We will concentrate on the probabilistic approach due to it decentralized nature.

Analysis: The Javelin system, like P^3, has a layered structure, with a fair amount of decentralization within the layers. Javelin, however, not only allows its hosts to directly interact with each other, but also lets them function in an autonomous fashion. The probabilistic scheduling approach requires hosts to get some tasks initially and host information from the broker layer, but lets these propagate among the hosts in an autonomous fashion thereafter, without further interference from the brokers. This satisfies most of the conditions of decentralization mentioned in Section 2.2 - by piggy-backing information about resources with tasks, hosts manage to discover additional resources in a decentralized fashion; hosts exhibit a fair amount of functional autonomy by "stealing jobs" directly from other hosts and interaction between hosts take place in a decentralized peer-to-peer fashion as well.

The broker layer works in a similar fashion too. A broker passes on a fraction of its current information to a new broker at the time of creation, and interacts with other brokers in a peer-to-peer fashion. The client-host relation, however, is controlled by the broker layer and all jobs initiate from a user to a broker, who then assigns it to some host. This leads to some amount of centralization in the system since the interaction between clients and hosts is indirect, controlled by the broker layer.

The benefits of decentralization are evident in Javelin. The responsibility of the brokers is reduced even further when compared to servers in XTremWeb or managers in P^3. Unlike XTremWeb, Javelin allows host machines to directly interact with each other and "steal jobs". Though P^3 allowed hosts to directly interact with each other and provided storage for fault tolerance, the associated decisions are always taken by managers. In comparison, brokers in Javelin are only responsible for passing the tasks initially to the host layer. Thereafter, hosts can autonomously find and perform these tasks by directly interacting with each other. This allows the host layer to achieve additional load balancing without further interference from the broker layer. Brokers do not interfere in fault tolerance as well. The clients achieves fault tolerance through Eager Scheduling. Javelin also allows brokers to dynamically create new brokers from existing hosts (secondary brokers) if needed. P^3 had similar options, but since Javelin reduces the responsibility of brokers even further, the need for creating additional brokers will be less, leaving more hosts available for computation.

To summarize, Javelin has a similar structure to P^3, but achieves a greater degree of decentralization by further reducing the role of the broker layer. However, the fact that all tasks reach a host initially through a broker creates a small amount of functional dependency, so we do not classify Javelin as a completely decentralized system.

We have taken a look at what we mean by an Autonomous Computing System and the various levels of autonomy that some of the existing distributed

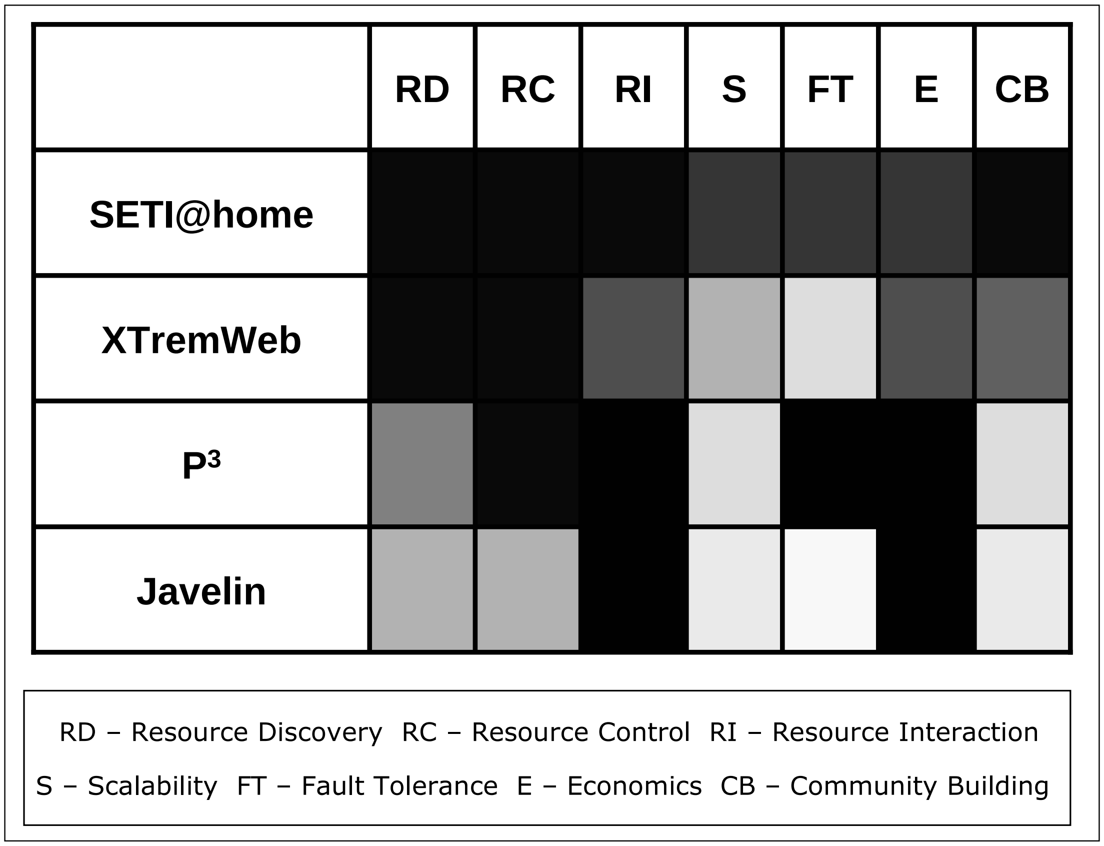


Figure 2.2: A comparison of the case studies in term of degree of autonomy they show and the corresponding benefits. The first three columns correspond to the three areas of autonomy that we mentioned. The remaining columns refer to the associated issues. A lighter color corresponds to greater autonomy and greater advantages. One can see that there appears to be a direct correlation between the degree of autonomy and the associated benefits.

systems support. Figure 2.2 gives a diagrammatic summary of our case studies, comparing their degrees of autonomy with the associated benefits. It can be observed that there seems to be a direct connection between autonomy and associated benefits.

We next present a detailed overview of the related work in this area so that we can have a better understanding of what the contributions of this dissertation are and how they differ from existing research.

Chapter 3

Resource Discovery

3.1 The Resource Discovery Problem

Before we describe the GUARD protocol [72] and how it manages to locate desired computational resources in an autonomous fashion it is important to understand what a computational resource is and how it is often different from pure data.

3.1.1 Available Resources in an ACS

One of the biggest advantages of using a large scale distributed system is the large volume of resources that it can provide the user with. Recent studies [57, 23, 1] have shown that a significant volume of computing resources stay idle for a considerable amount of time. A user is thus no longer limited to using his own computing resource but should "ideally" be able to use the strength of the millions of idle machines connected over the Internet (Appendix A takes a look at how powerful some of these isolated nodes are from a computing perspective). It is not difficult to imagine the World Wide Web as a large supercomputer where users can submit jobs and share resources. For our work we are concerned with computing resources as opposed to just data (e.g. MP3s, video files etc.).

We define a computing resource as *a resource that is used or required for a computational task to run to completion.* There are several computational resources - processing power, memory, storage space, network bandwidth and available software are just some computational resources. Each node in the system comprises one or more of these resources. Resources have a set of possible values that they can take, where the *value* of a resource is a parameter that a user might desire (e.g. “Does this node have 256MB of free memory?” - here “free memory” is a resource and 256MB is the value associated with it). Based on values, one can categorize all computing resources into two broad categories:

- **Permanent Resource**: These are resources whose value doesn’t change with time. Examples of permanent resource are architecture of processors, installed software, presence of desired hardware etc. These resources can thus be tracked by using a boolean value that denotes if they are present or not. In other words, the answer a node returns for a query of this resource, doesn’t change with time.
- **Dynamic Resource**: These are resources whose value changes with time. Examples of this would be processor utilization, free memory, number of active processes etc. where the same node can satisfy or fail the query for a certain resource at different times.

3.1.2 Locating Computing Resources

A user might be interested in a node (or collection of nodes) meeting one or multiple resource requirements (e.g. find a node that has Java installed and has a CPU utilization less than 25%). In a centralized system, the server(s) can query the participating nodes periodically and keep track of the values of the various properties associated with a resource. Users can then directly query the server for a desired node. In a decentralized scenario, nodes are only aware of their immediate neighbors and can only communicate with them. It is therefore more difficult to locate a desired resource in this scenario.

A computing resource is quite different from a data-centric resource and hence the vast volume of research addressing the problem of decentralized data-centric routing [55, 98, 88] cannot be directly applied to this scenario. Some of the properties that make the computing resources different from their data-centric counterparts include:

- **Dynamic Nature**: As mentioned earlier, many of the properties of computing resources like CPU utilization, amount of free memory etc. change with time. Thus one cannot hash their information to unique locations [98, 88] from where one can retrieve their information.

- **Perishable**: Many computing resources are perishable in nature i.e. they get used up. For example, a node having 1GB of free memory, if located may get used up and can no longer be used by other users seeking that property. It therefore doesn't make sense creating a one time index for these nodes because they can often not be used by more than user.

- **Non-replicable**: Data can be replicated and hence one can eliminate performance bottlenecks by replicating a highly accessed data in several places. Many computing resources do not support the notion of replication (e.g. how do you recreate a node that has Pentium IV and 4GB disk storage in a different place?).

- **Smaller Sample Space**: Unlike data, which has a huge number of variants, most computing demands are similar (or at least fall under a smaller set of categories). One can therefore make use of this empirical knowledge of user requirements more effectively in a computing scenario.

Having seen what computing resources are and how they differ from their data-centric counterparts we will now study in detail the GUARD protocol and how it can be used to locate computing resources in an autonomous fashion.

3.2 The GUARD Protocol

We first describe the basic GUARD protocol, whose goal is to locate an instance of a given resource in a large heterogeneous system without the use of centralized control or information. We also derive an analysis of GUARD's performance, and describe how the protocol can be extended to locate resources that satisfy a set of properties simultaneously.

3.2.1 The Basic GUARD Protocol

GUARD uses distance vectors [81] to maintain *likely* distance from resources. Unlike the use of distance vectors in routing IP addresses, where the target of a node is a unique node, there are likely to be *multiple* resources satisfying a request in a large distributed system. GUARD maintains the distance from the closest node having a particular resource. This information is updated via "gossiping" (nodes exchange information with their neighbors periodically) to reflect the consumption/addition/deletion of resources in the system. Nodes in the system only interact with their immediate neighbors, and do not communicate directly with any other node.

We will first introduce the various terms used in describing the protocol. We assume that the underlying system is represented as a graph $G = (V, E)$ with $N = |V|$ nodes. We shall initially assume that there are K types of trackable resources $(R_1, R_2, ..., R_K)$ in the system, but later show how GUARD can be used for resources, such as memory, that can be requested in different sizes. Nodes make requests for one of the K types of resources (Section 3.2.3 shows how GUARD can handle multidimensional requests) and the protocol tries to locate the closest node that satisfies the request.

The way GUARD works is simple. Each node in the system maintains a table of size K to track down the K different resources (we show in Section 3.2.3 how the table size can be reduced). The entries in the table are of the form $< D_i$,

$nbr_i >$ $(i = 1, 2, ..., K)$, where D_i is the node's current knowledge on the number of hops that need to be traveled to reach a resource of type R_i, and nbr_i is the neighbor with the least value of D_i. Initially all nodes have their tables initialized with all D_i's set to ∞. If a node itself has any of the resources R_i, it sets the value of D_i in its table to 0. Periodically nodes *gossip* by sending a copy of its table of D_i's to its neighbors. Whenever a received value D_i' from neighbor k, such that $D_i' + 1 < D_i$, the node updates the entry to $< D_i' + 1, k >$.

On receiving a request for R_i, a node checks if its value of D_i is 0 (implying it has the resource), in which case it services the request and sets the value of D_i to ∞ (similarly a node resets the value of D_i to 0 once the resource is released). If the node doesn't have the resource it forwards the request to nbr_i. A request that is forwarded more than TTL hops is killed. A node also kills a request in case a node does not have a resource and has no neighbor it can route the message to ($D_i = \infty$) (the node can wait for a while and re-try to route the message but in our implementation we haven't done this).

The detailed algorithms for gossiping (updating) and routing of requests in GUARD is shown in Figures 3.1 and 3.2.

3.2.2 Analysis of the Protocol

GUARD uses the distance vector approach that has been widely studied in the networking community. While this approach is unscalable for its traditional use of locating *unique* nodes in an Internet-like scenario, it is extremely well suited for our *resource location* problem when the number of resources that users search for is fairly limited. Also, there are likely to be multiple nodes that might satisfy a particular request. This makes it easier to propagate resource information in the system (since updates in a node's status will only affect those nodes that were using it as a closest resource source) and the problem often reduces from locating a unique node (routing in the Internet) to locating one of several satisfying nodes. This makes the distance vector approach more conducive to the dynamic

```
// Assume the node has N neighbors
// Assume there are K different types of resources in the system {$R_1$, $R_2$, …, $R_K$}
// Each node maintains a K element table where each entry of the table is of the form <$D_i$, $nbr_i$>
// where $D_i$ is the distance in number of hops from a node having $R_i$ and
// $nbr_i$ is the neighbor where requests for $R_i$ should be forwarded

void gossip() {        // Gossiping protocol performed periodically
    // First update own information
    for (i = 1 to K) {
        if (node contains $R_i$) {
            $D_i$ = 0; $nbr_i$ = self;
        }
    }
    // Communicate with neighbors
    for (i = 1 to N) {
        for (j = 1 to K) {
            if ( neighbor i’s $D_j$ < ($D_j$-1)) {
                $D_j$ = neighbor i’s $D_j$+1; $nbr_j$ = i;
            }
        }
    }
}
```

Figure 3.1: The gossiping protocol that nodes in the system using GUARD periodically follow. Each node has a routing table that maintains the shortest distance to a specified resource and the neighbor it must route messages for the resource to.

```
// Assume that each node has a unique identifier – id of type Id
// Assume that a query Q comprises of the requested resource type $R_X$ and the number of
    hops the message has traveled

Id route(Query Q) {  // Returns the id of the closest node having Q.$R_X$
    // Check if node satisfies the request
    if($D_X$ == 0) // node has $R_X$
        return id; // returns id of itself

    // Check if number of hops has exceeded maximum limit TTL
    if(Q.hops == TTL)
        return error;

    // Pass on message to corresponding neighbor after updating number of hops
    Q.hops++;
    N = $nbr_X$;
    return  N.route(Q);
}
```

Figure 3.2: The routing protocol that nodes in the system using the GUARD protocol use to locate a resource

computing resource location problem.

We now give a theoretical estimate of GUARD's performance. The estimate is an upper bound that allows users to determine whether such an approach is suited for their system and/or to tweak the protocol's parameters to meet their requirements.

We begin by introducing the terms used in our analysis. Let F_u be the frequency at which nodes gossip about their information and F_r be the frequency at which requests are made in the system. Resources are borrowed for an average time length of L, i.e. a resource once allocated remains unusable on average for L time. P_r denotes the percentage of nodes in the system that initially have a resource of size r and are willing to share it.

Suppose a request for resource r ($r \epsilon \{R_1, R_2, ..., R_K\}$) originates in the requesting node, n_r, at time T. Let n_d be the node we expect to be the final destination of the query and l_{n_r} be the distance (in number of hops) between n_r

and n_d, i.e. l_{n_r} is the value of D corresponding to resource r in n_r's routing table. For our derivation, we assume the latency of requests is negligible compared to the frequency of gossiping. We also make the uniformity assumption that resources and requests are distributed among all the nodes in a way that makes each resource be the "closest" resource to about an equal number of requests. While this is not true for all topologies, it may hold for important scenarios including sensor-like graphs and existing peer-to-peer systems like CHORD [98] (and it makes our analysis possible).

Let $P(r, T, n_r)$ = probability of failure of request for r at time T, starting at node n_r.

$\leq$ probability of r being taken in n_d in time $T - \frac{l_{n_r}}{F_u} \leq t \leq T$

- The $\leq$ inequality is used because the request might be satisfied by a different node.

$\leq$ probability that **any** request for r is serviced by n_d in time $T - \frac{l_{n_r}}{F_u} \leq t \leq T$

$\simeq$ 1 - $e^{\frac{-l_{n_r} \cdot F_r}{F_u \cdot N_{r_T}}}$, where N_{r_T} is the number of nodes having resource r at time T.

- By our uniformity assumption, the number of other requests that are issued in the time period $T - \frac{l_{n_r}}{F_u} \leq t \leq T$ and which are closest to node n_d is Poisson distributed with mean $\frac{l_{n_r}}{F_u} . \frac{F_r}{N_{r_T}}$. The above expression is 1 - (probability of **no** such request).

Now assuming we want the failure rate to be below a given fraction f_F we can write

1 - $e^{\frac{-l_{n_r} F_r}{F_u N_{r_T}}} \leq f_F$

$\Rightarrow e^{\frac{-l_{n_r} F_r}{F_u N_{r_T}}} \geq$ (1-f_F)

$\Rightarrow e^{\frac{l_{n_r} F_r}{F_u N_{r_T}}} \leq \frac{1}{1-f_F}$

Now taking natural logarithm on both sides and simplifying we get

$$\boxed{F_u \geq \frac{l_{n_r} F_r}{|ln(1-f_F)| N_{r_T}}}$$

Now both l_{n_r} and N_{r_T} depend on several factors like the topology of the network, the values of P_r, L etc. but can be estimated, given a particular scenario. This implies that given a particular setup one can simply tweak the gossiping frequency to achieve a desired rate of success. We verify these predictions in Figure 3.13.

3.2.3 GUARD for Multi-dimensional Queries

The GUARD protocol that we described is capable of locating nodes having a specific resource. However, in a realistic scenario the application is likely to need resources meeting *multiple* criteria (e.g. a Pentium IV processor with 1 GB memory running Linux). We shall call these requests *multi-dimensional* requests [97]. While we can use the already described version of GUARD to locate individual nodes meeting these requirements, there is no guarantee that these resources will be in the same node. Moreover, the closest distance to nodes meeting individual requirements might be different from nodes meeting all these requirements (e.g. The closest node having 1GB RAM might be 3 hops away and the closest node having a Linux OS might be 2 hops away, but the closest node having both these properties might be 5 hops away).

One way to tackle this problem would be to track all possible request types. While this would solve the problem, it is impractical since even a small number of trackable properties can lead to a huge number of possible requests. We therefore consider a variant of the GUARD protocol, similar to a technique used for data-centric systems in [50], that populates the node tables with the most popular queries.

Assume that there are K different types of resources $< R_1, R_2, ..., R_K >$ (for simplicity we assume that a resource is Boolean, i.e. $R_i = 1$ if a node has the resource, otherwise it is 0). A user may want to locate a node having one or more of the K resource types. We use the following approach. Each node has a table of fixed size, but now the table entries are of type <*request_type,distance,nbr,count*> where *request_type* defines the set of types constituting the request, *distance* is the number of hops from the resource, *nbr* refers to the neighbor of the node to forward requests for this combination of resources, and *count* keeps track of how many times this particular resources type has been requested. Initially all the table entries are initialized with a single entry, with all fields corresponding to the resources the node owns set to 1 in the entry.

Whenever a node gets a request, it makes a table entry for the request (if it doesn't already have one), or increments the entry's *count* (if it does). Since the table is of finite size, an entry with the minimal value of *count* is evicted[1]. If the node can satisfy the request itself, it does so. If its table has an entry that can satisfy the request or a superset, it forwards the request to the indicated node. Otherwise, it sets the distance in the table to infinity, and the request fails. Via gossiping, table entries are propagated to neighbors and beyond. Eventually, a node is found with the resource and the gossiping results in table entries with finite distances.

The above modification to the protocol maintains the simplicity and autonomous behavior of GUARD while addressing the issue of multi-dimensional requests. The table entries get populated with the more commonly issued request types.

3.2.4 GUARD for Non-boolean Resources

Some resources, such as memory, are not Boolean but come in different sizes. Any request for a particular size can be satisfied with a resource of greater size. In this case, GUARD has a finite set of Boolean requestable sizes (e.g. 64, 128, ..., 2048 MB). We used a "best fit" strategy, where if a node cannot satisfy a request itself, it forwards it to the node in its table that has the smallest acceptable value (even if that node is further away than one with a larger resource). Other strategies, such as "closest acceptable node", are possible and worth investigating.

[1]One can try out other cache eviction policies too. e.g. Least Recently Used etc.

3.3 Experimental Results

In order to test the effectiveness of GUARD we built a simulator (a discrete event simulator that was built from scratch) that allowed us to experiment with several aspects of the protocol. We tested the protocol for two sets of topologies - (i) sensor-like[2] networks (SN) where we randomly scattered the nodes over a square grid and then connected all nodes within a given radius, and (ii) tree-shaped networks (TN) that reflect the hierarchical topology of networked clusters. Each experiment had a *running* time and a *startup* time. The startup time was provided to have the system reach "steady state". For all experiments the startup time was fixed at 1000 seconds of simulation time. This value was chosen as sufficient in our experiments for the performance to stabilize to a steady or periodic behavior. The running time, after startup, was set to 10,000 seconds.

We begin by choosing plausible values of the parameters for our initial set of experiments. (In Section 3.3.2 we will show how some of these parameters affect the performance of GUARD.) The requests were distributed randomly across the lifetime of the experiments. Resources once claimed remained with the requesting node (the duration of a task was distributed randomly between 15-30 minutes i.e. 900-1800 seconds). Not all nodes in the system shared their resources (to make our simulator more realistic) and the percentage of nodes that shared their resources p_s was set to 70% for the initial experiments. Nodes updated their information once every 50 seconds. For the initial experiments we considered the resource to come in six different sizes. Each of the nodes sharing their resource with others had a resource of one of these sizes with the probabilities {*.05,.1,.3,.3,.15,.1*} and the requests for these resources had a probability of {*.05,.1,.2,.35,.15,.15*} (chosen to make some of the less available resources more heavily requested). For the experiments shown in Figures 3.5 and 3.8 we used variable sized resources where a first-fit approach was adopted for answering the queries, i.e. a request was routed

[2]We anticipate that one of the major uses of large scale decentralized computing will be sensor networks.

to the closest node whose resource size was greater than or equal to the that of the request. For all other experiments we used an exact-fit approach.

3.3.1 Performance of Guard

The main test for a protocol is its performance. We tested the performance of GUARD by varying several parameters to test its efficiency and feasibility. To test the basic protocol we created systems of size $N = 1000$, 2000, 4000, 8000, 16000 and 32000 for both the topologies *SN* and *TN*. For each experiment, the number of requests was $2N$ and the requesting nodes were chosen randomly from all the nodes. We compared GUARD to three other protocols used for similar purposes described in [47], namely:

- **RAND**: In this protocol nodes randomly routes the request to one of its neighbors until it is successfully answered or it reaches a maximum distance TTL (for our experiments we set $TTL = 100$).
- **HIST**: Nodes keep a history corresponding to each resource type, tracking the neighbor that successfully routed it the last time. Requests for a resource are routed for a maximum of TTL times by this method, thereafter, RAND is used for up to an additional TTL steps. On successful completion of a request, all nodes on the path updated the entries to show the successful route.
- **FREQ**: Nodes keep track of the number of requests (irrespective of the resource type) that each neighbor answered. Requests are routed to the most successful neighbor. Like HIST, on failure the protocol resorts to RAND.

The results for Boolean resources are shown in Figures 3.3 to 3.8. We evaluated performance based on two criteria - (i) the percentage of requests that were successfully answered (a failure means there existed a resource of a particular

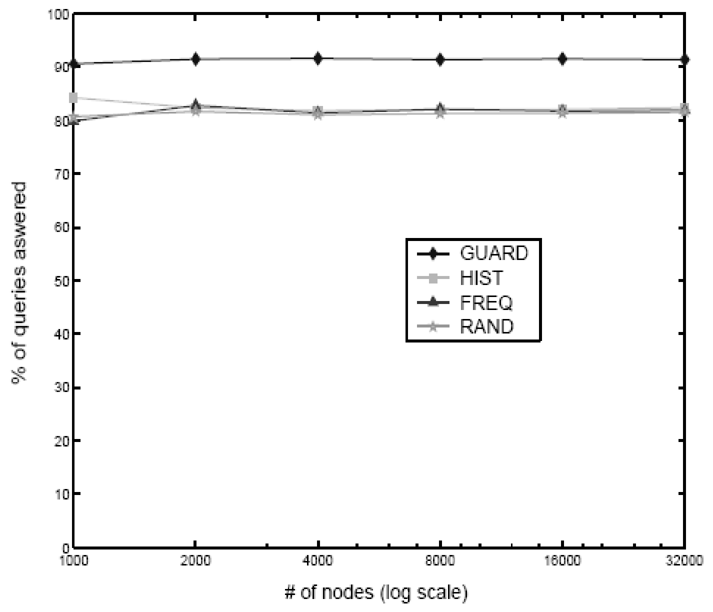


Figure 3.3: Percentage of queries successfully answered by the various strategies for the Sensor-like Network (SN)

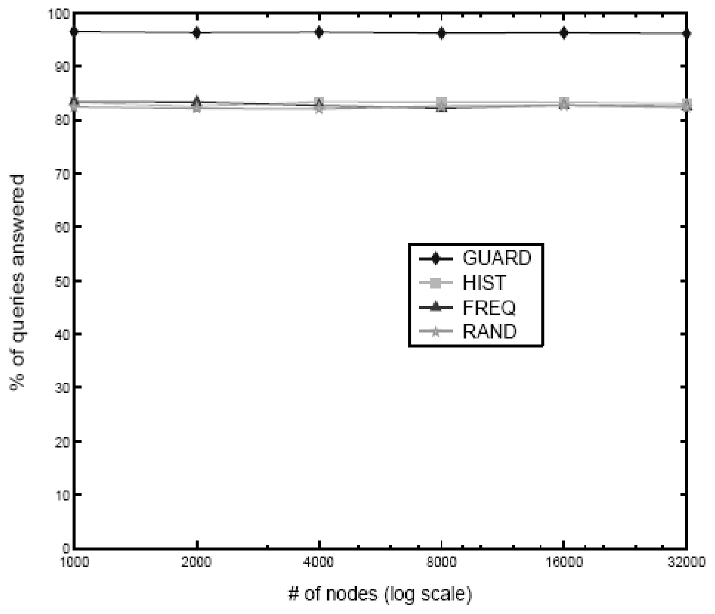


Figure 3.4: Percentage of queries successfully answered by the various strategies for the Tree-like Network (TN)

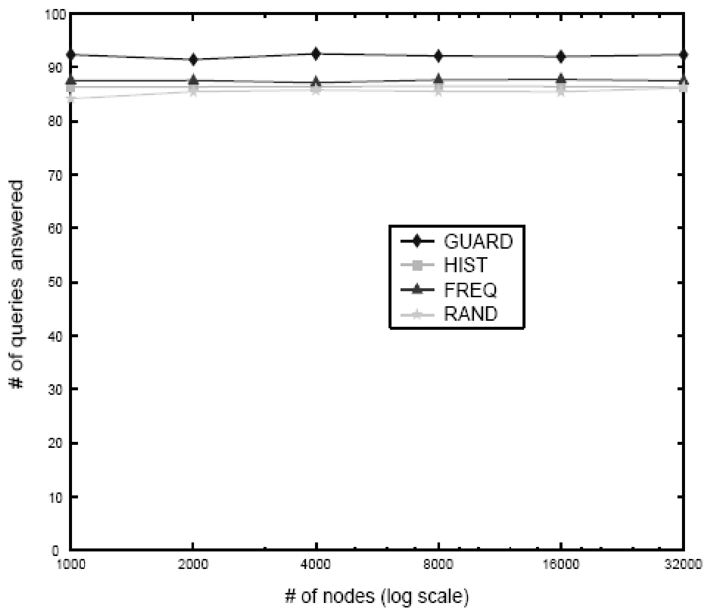


Figure 3.5: Percentage of queries successfully answered by the various strategies when a first-fit approach is tried instead of a best-fit policy.

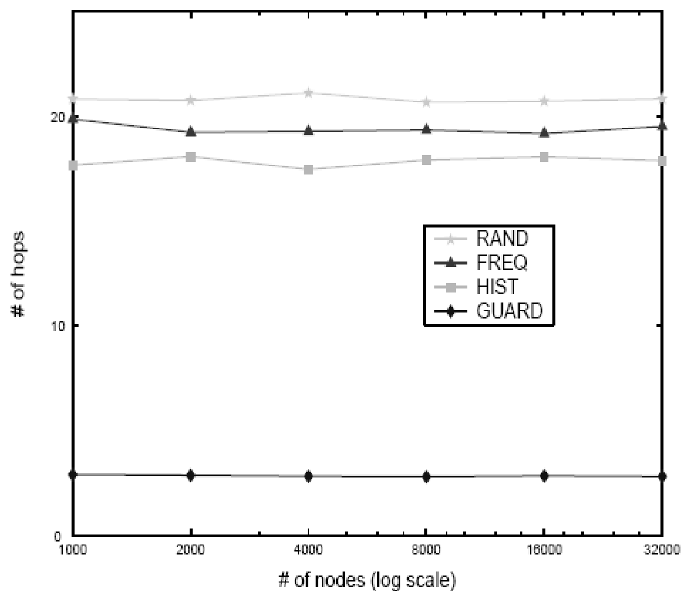


Figure 3.6: Average number of hops needed by the various strategies for the Sensor-like Network (SN)

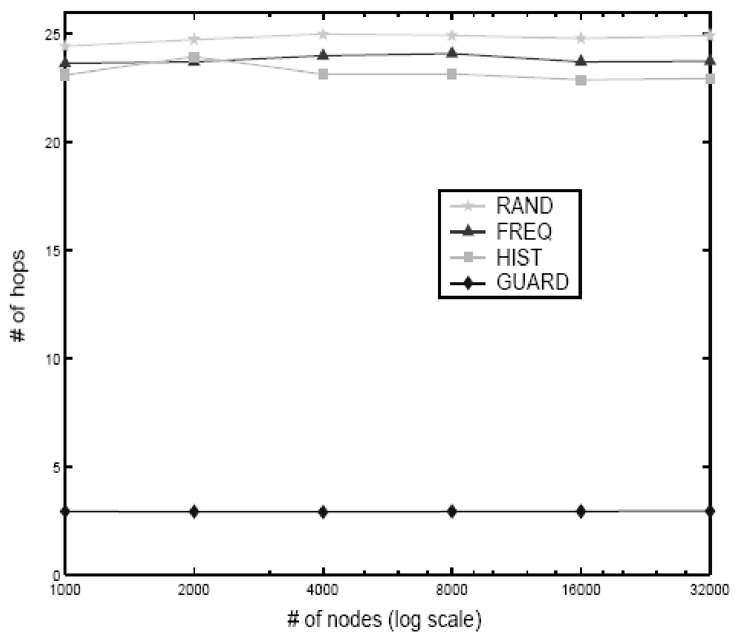


Figure 3.7: Average number of hops needed by the various strategies for the Tree-like Network (TN)

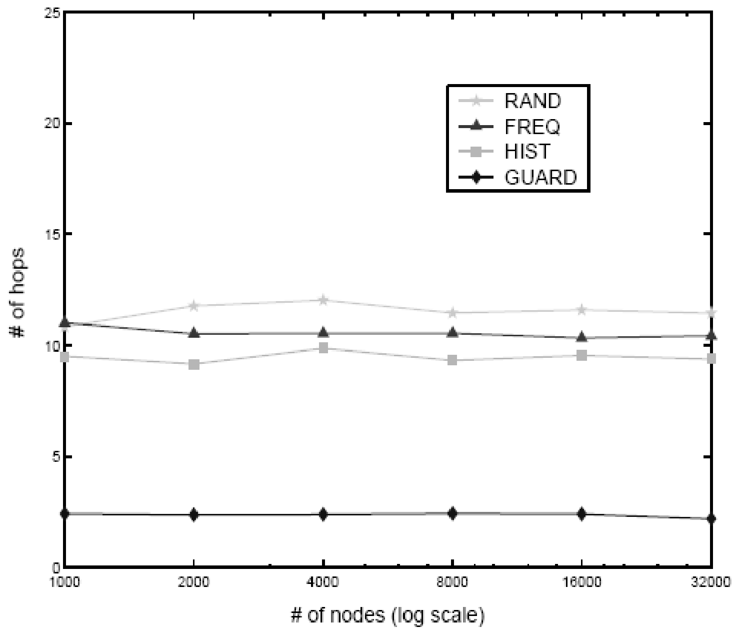


Figure 3.8: Average number of hops needed by the various strategies when a first-fit approach is tried instead of a best-fit policy.

size but the protocol failed to locate it; if there is no such resource, it is not counted either way) and (ii) the average number of hops taken to reach the resource.

One can see that GUARD significantly outperforms the other protocols in the percentage of requests answered, and it always uses a smaller number of hops. GUARD is scalable and performs well for both topologies and all the graph sizes. Intuitively, this improvement comes largely because the distance vector approach helps to identify the *nearest* resource while the other protocols target "any resource". Though GUARD spends effort in periodically updating its information, this effort helps in achieving significant benefits. It must also be mentioned that the small value for the average number of hops needed by GUARD does not mean it wasn't suitable for locating distant resources. In our experiments GUARD worked successfully even when the nearest resource was 30-40 hops away.

The results in Figures 3.5 and 3.8 use variable-sized resources. One can see that even though the performance of RAND, HIST and FREQ improves both in terms of percentages of requests answered and number of hops taken to locate the resources, GUARD still consistently outperforms these heuristics. This suggests that GUARD can be used to locate both resources having a Boolean value (like OS, software, architecture etc.) or resources that can satisfy a range of values (memory, storage etc.)[3].

3.3.2 Effect of Various Parameters on GUARD

To test the effect of various parameters on GUARD's performance, we used the setup of the previous experiment for N=4000 and varied one of the parameters, keeping the others constant. The parameters we studied were (i) the percentage of nodes sharing their resources, (ii) the inter-update time at which nodes update their information, (iii) the number of requests made and (iv) the average duration of a task. We conducted the experiments for both topologies and

[3]The dividing of resources like memory, storage etc. into smaller categories can lead to fragmentation of the resource and has not been dealt with in our work.

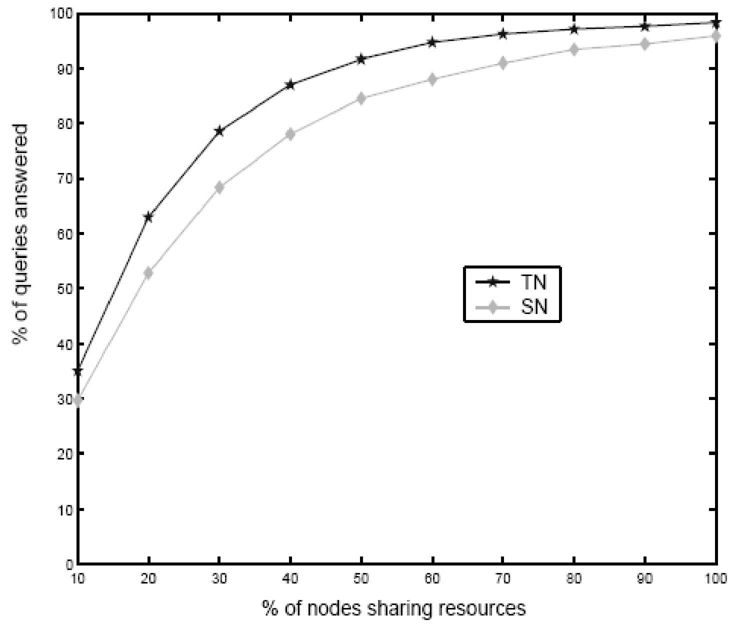


Figure 3.9: Effect of the percentage of nodes that share their resources on the percentage of queries answered by GUARD

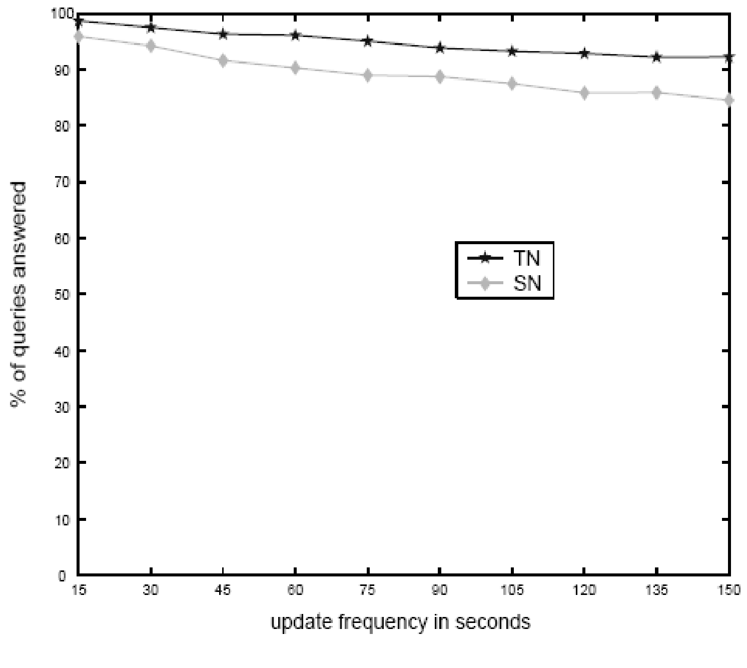


Figure 3.10: Effect of the update (gossip) frequency on the percentage of queries answered by GUARD

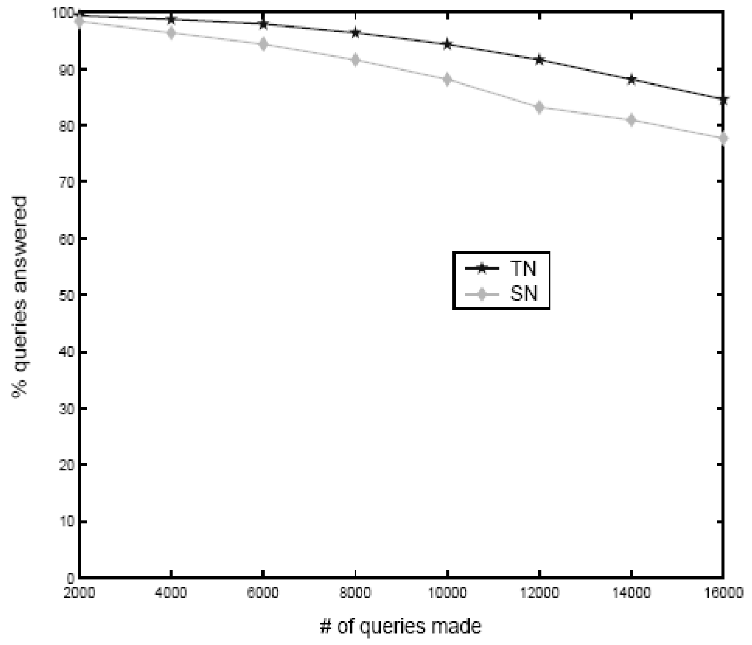


Figure 3.11: Effect of the number of queries made on the percentage of queries answered by GUARD

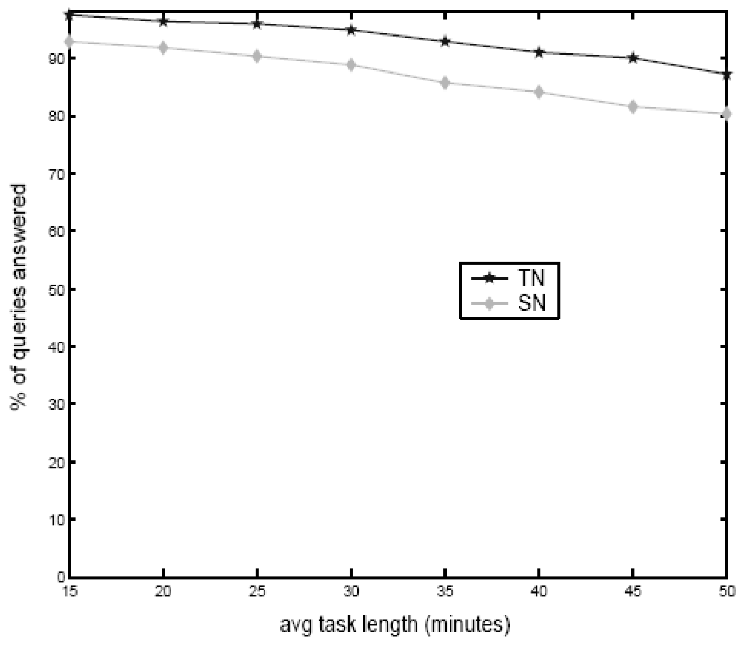


Figure 3.12: Effect of the average task duration (mean task length) on the percentage of queries answered by GUARD

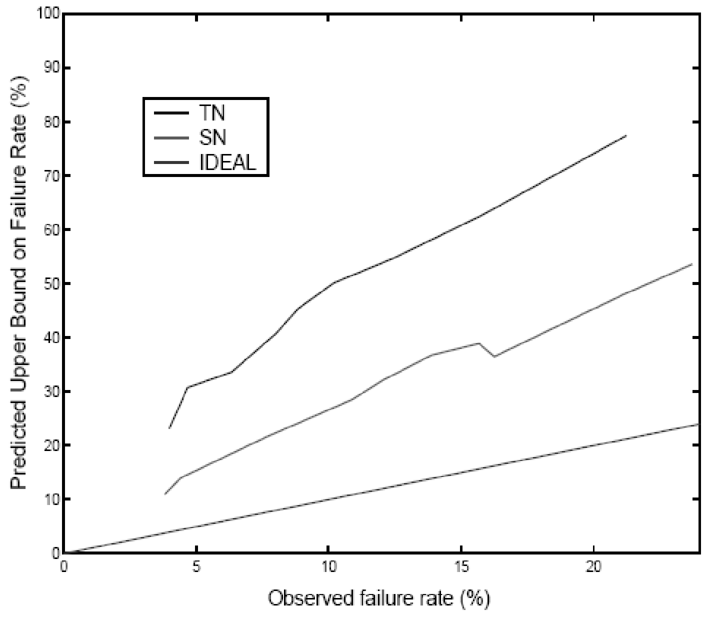


Figure 3.13: Comparison of upper bound on failure provided by our analysis vs. actual failure rate of GUARD. Observe that the actual failure is always below the predicted value.

the results are shown in Figures 3.9 to 3.12. The results give an idea to how the various parameters affect performance and support the analytical model presented in Section 3.2.2.

3.3.3 GUARD: Observed Performance Vs. Predicted Performance

To test the validity of the analysis presented in Section 3.2.2, we calculated the failure rate of the experiments mentioned above and compared it to the upper bound provided by our analysis. The value of N_{r_T} was calculated by the simulator using the actual number of resources of a size that was left in the system[4]. We have provided the results for both the topologies and have also provided the ideal curve (where the predicted rate equals the observed rate) for comparison

[4]This might be difficult to do in an off-line context but can be estimated using an expected steady state value based on the rate of requests and the rate of release of resources.

in Figure 3.13.

We see that the observed failure was always bound by the upper bound that our analysis provided. This shows that the analysis does indeed provide an upper bound that can be used to determine a desired value of F_u. It can also be seen that the predicted rates are almost a constant factor higher than the observed rates (this factor varies for the two topologies). We conjecture this is due to the possibilities of multiple paths to destination nodes and the possibility of multiple destination nodes. Moreover, the l_{n_r} term appearing in the exponent is representative of the worst case scenario. In case one desires a tighter upper bound they can use the average value of $\frac{l_{n_r}+1}{2}$ for the analysis instead.

3.3.4 Revised GUARD (Multi-dimensional Queries)

We now evaluate the performance of the revised GUARD protocol (mentioned in Section 3.2.3)where nodes can request a combination of the K resources and the lookup table has a limited size. Like the previous experiments each node had a 70% probability of having a resource. All combination of requests for a given number of resources occurred with equal probability. We tried experiments with K=5 and 6. For K=5 the requests comprising of 1-5 resources appeared with probabilities of $\{$*.35,.28,.21,.11,.05*$\}$. For K=6 these probabilities were $\{$*.30,.27,.22,.08,.08,.05*$\}$. We ran experiments for both the topologies by varying the size of lookup table (note: for size=2^K the protocol reduces to the standard version of GUARD, except for the startup time taken to propagate the requests). The results are given in Figures 3.14 and 3.15.

We observe that the modified version of GUARD performs well: Even for small sizes of the table, GUARD manages to answer more than 80% of the queries successfully. This is comparable to the performance of the other protocols observed earlier in terms of the success rate, while GUARD uses many fewer hops to answer these requests[5].

[5]HIST, which was the second most efficient in terms of number of hops is likely to run into

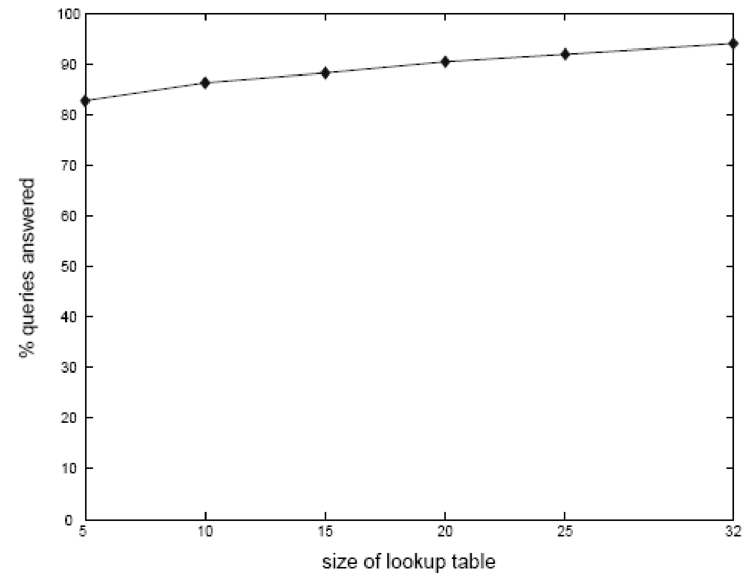


Figure 3.14: Performance of the revised GUARD protocol that allows multi-dimensional queries for a combination of 5 different types of resources

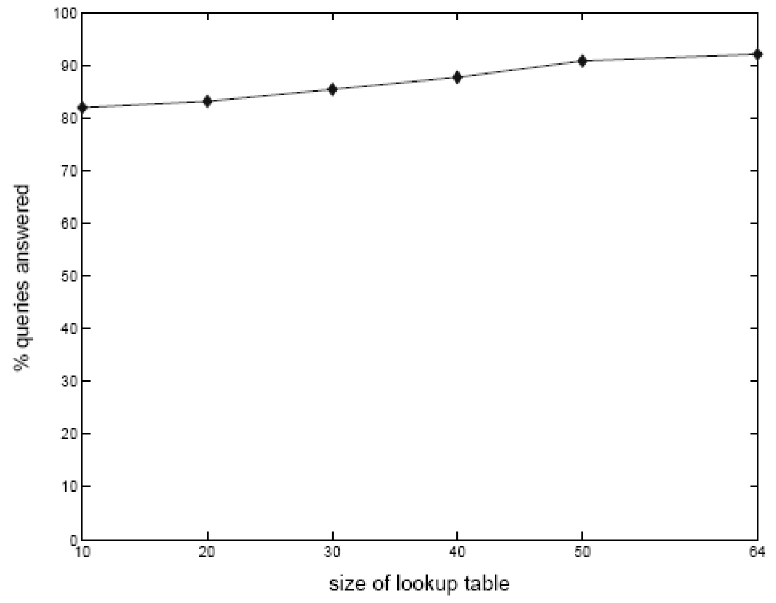


Figure 3.15: Performance of the revised GUARD protocol that allows multi-dimensional queries for a combination of 6 different types of resources

3.4 Practical Issues Associated with GUARD

The basic GUARD protocol and its multi-dimensional variant can be further modified to meet certain additional situations. In this section we discuss some of the practical issues that we did not discuss associated with GUARD.

3.4.1 Propagation of Stale Information

One of the issues associated with a distance vector based approach is that it is slow in propagating information of staleness i.e. when a resource is actually consumed, it takes a long time for that information to propagate through the system. This happens because nodes interact only with their neighbors. Take the example shown in Figure 3.16. Now assume that N_1 has some resource and hence the corresponding distance counters of N_2 and N_2 are 1 and 2 respectively. Now if the resource in N_1 gets consumed then ideally N_2 and N_3 should reflect it immediately but in reality both these nodes will keep reading incorrect information from each other and slow down the convergence significantly.

We address this issue by associating 2 additional fields with each routing table entry, the source (*src*) and time-stamp (*ts*) of the associated resource. The *src* tag contains the unique *id* of the node where the resource actually exists and the *ts* field is a time-stamp denoting when this information was propagated. This time-stamp doesn't have to be a system wide synchronized time value but is local to each node. When a node sends its current routing table information to its neighbors, it updates the field corresponding to any resource it owns, with the current value of the time-stamp. If a neighbor realizes that the node is closer to a resource than itself, it checks if they both depend on the same *src* value. If the source values match, the node checks if the value of *ts* is a more recent one. In other words, nodes only update their resource distance from a source if it is a more recent value.

the same problem of a limited size lookup table.

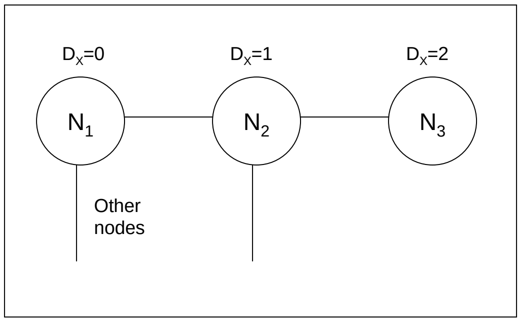


Figure 3.16: Example showing the convergence problem. Imagine now that some other node uses up D_X in N_1. N_2 and N_3 will then keep passing on incorrect information to each other, slowing down the convergence rate significantly.

3.4.2 Dealing with Failures

It is possible in GUARD for a node to believe that one of its neighbors has a resource when it actually doesn't. This happens due to the delay between table updates. In our experiments whenever such a situation arose the request was dropped and reported as a failure. However, in a practical implementation the node can wait for a while before continuing the routing process. The time to wait can either be a constant time or proportional to the frequency of updates and the number of hops the message has already traversed.

3.4.3 Dealing with Multi-node Requests

Unlike SWORD [79, 80], the current version of GUARD does not support requests asking for a pool of nodes. GUARD returns to the user the nearest node satisfying a given requirement. It doesn't specify anything about the handshaking protocol between the two nodes thereafter. One can therefore easily implement a resource grouping/pooling protocol on top of GUARD by making multiple resource requests and booking them[6].

[6]However, one has to ensure that resources are freed in case all requirements of a pool of nodes is not met, avoiding unnecessary starvation.

In this chapter we studied the GUARD protocol. GUARD uses gossiping with neighbors to propagate distance information of resources. GUARD is completely autonomous and each node only interacts with its immediate neighbors in the underlying topology. We saw that GUARD significantly outperformed other decentralized techniques. GUARD also supported multi-dimensional queries.

Having studied how computing resources can be detected in an autonomous fashion, we now discuss the issue of using these resources effectively for autonomous task scheduling.

Portions of the text of this chapter are a reprint of the material as it appears in [72]. The dissertation author was the primary researcher and author and the co-authors listed on this publication directed and supervised the research which forms the basis of this chapter.

Chapter 4

Autonomous Task Scheduling

Task scheduling has been one of the most extensively studied areas in distributed computing. The purpose of a distributed computing system is to use its available resources to facilitate computation. In order to achieve this objective it is extremely important to schedule tasks efficiently to the right resource(s). Ideally, in centralized systems, a dedicated scheduler is responsible for mapping tasks to resources [48, 5, 21, 87, 32, 62, 65, 44, 101]. The scheduler is normally aware of the number of resources, their availability, their computing ability and the capacity and speed of the interconnecting network(s). The scheduler also often has a good knowledge of the tasks and their requirements. This comprehensive knowledge often allows the scheduler to do a good job of mapping a task (or group of tasks) to a resource (or pool of resources).

However, in an autonomous scenario, there are no centralized schedulers where nodes can submit their tasks. Thus the job of task scheduling is delegated across all nodes in the system. Each node has to make its scheduling decision locally, while ensuring that they are in the best interest of the overall system. This is the problem that we address in this chapter.

We deal with homogeneous and independent tasks (other distributed computing efforts have used a similar task model [91, 68]). All tasks are submitted by a single user/node (we later discuss in Section 4.4.3 how one can easily extend our solution to multiple users). This node has a huge pool of homogeneous tasks that is submitted locally to one (or more) of the neighbors. A task, in our system, can either get performed by the node, or get passed on along the system to its neighbors. This process continues until some node executes the task. *Performance is measured in terms of steady state throughput* and not total completion time since we assume that the submitting node has an unlimited pool of tasks.

This chapter presents A-FAST (Autonomous Flow Approach to Scheduling Tasks) [71], a protocol that provides an autonomous solution to the above described problem. The autonomic behavior of fluid networks, using pressure as a guiding force, forms the key inspiration for our work. One can imagine the nodes in a grid as fluid reservoirs, and the links as pipes connecting these reservoirs. Tasks are analogous to the circulating fluid in this scenario. In case of the fluids, pressure helps in bringing the system to a steady state without the use of any centralized control. We propose a similar approach where nodes autonomously measure their own pressure. This pressure is then used to decide when to move a task to a neighboring node, eliminating the need for centralized control over scheduling. A-FAST shares similarities with well-known techniques like Cycle Stealing [13] and RID [69], but differs from these techniques by taking both computation and communication into account, which makes it better suited for a wider range of networks. We show how several important scheduling-related issues, including fairness, throughput and reliability, can be easily incorporated in our approach.

We begin with a formal description of the problem and then describe the A-FAST protocol in detail. We then show how A-FAST can be used to achieve high performance and also achieve greater reliability. Simulation results are then provided that experimentally evaluate various aspects of the A-FAST. We also present some real world experimental results and the lessons we learned from them.

4.1 Problem Definition

We begin with a formal description of the problem. We are given a labeled, directed graph G = *(N, E, P, C)* representing the network. The nodes of G is the set $N = \{0, 1, ..., n-1\}$, with each node representing a computing resource (processor, computer, cluster etc.) Each node i ($i \epsilon N$) has a computing speed represented by $P(i)$ ($P : N \rightarrow R^{+}$), denoting the number of tasks the node can complete in a unit time. $E = \{(i, j) : i, j \epsilon N\}$ represents the set of edges (links) connecting the various nodes in this graph, and $C(i, j)$ denotes the number of tasks that can be sent from node i to node j in a unit time i.e. $C : N \times N \rightarrow R^{+}$. All tasks are of equal size (both in computation and communication)[1] and initially reside in the source node 0. The graph G is dynamic in nature, i.e. *(N, E, P, C)* can change during execution. Nodes and edges can be added to or deleted from N and E (except for node 0, which is always present) and the functions $P(i)$ and $C(i, j)$ can also change. Our objective is to maximize the overall throughput of the network i.e. maximize the number of tasks completed per unit time.

4.2 The A-FAST Scheduling Protocol

The A-FAST protocol exploits the fact that incoming tasks can be buffered in a node. The protocol is divided into three parts - task receiving, task sending and task processing. Nodes begin by advertising their current *pressure* (p) to their immediate neighbors, requesting them for tasks. On receiving a request, a node compares the requester's p to its own to decide whether the request should be serviced. Such an approach allows us to do away with the need for a centralized scheduler, and instead make all scheduling decisions locally based on differences in pressure, and yet develop a system-wide notion of need and equilibrium. If a

[1]Although we have not yet performed the experiments, we conjecture that if tasks are of different sizes, but have a constant computation-to-communication ratio, that the behavior of algorithms will be similar to the equal-size task problem. An interesting open question is how to make scheduling decisions when the ratios are different but known.

node does not service a request, it informs the requestor of its decision. On being serviced by a neighbor, a node requests another task. However, if its request is denied, it waits for a set length of time before making another request. Nodes thus periodically query their neighbors, requesting further tasks. To process a task, a node takes a task from its buffer. If the buffer is empty the node waits till it receives a task.

In its simplest form A-FAST tries to mimic fluid networks - transferring tasks from locations of lesser need to those of greater need. There are however several differences between the two mediums, and subsequent sections will explain how we address these issues.

4.2.1 Scheduling in Dynamic Heterogeneous Environments

We now show how A-FAST can be used to schedule tasks in a heterogeneous system. We achieve this by defining the pressure, p_i of node i, to be simply the number of outstanding tasks in its *task buffer*, TB_i. For each edge $(i, j) \epsilon E$, A-FAST makes use of an *intermediate buffer*, IB_{ij}, where IB_{ij} is a buffer on node j that holds responses sent to it from node i. TB_i has a capacity of m_i "*slots*", where each slot can hold one task. Each of the slots in TB_i is in one of the following states:

- **S1**: the slot is "empty".
- **S2**: a task is being transferred into the slot from one of the IB_{ji}s.
- **S3**: the task in the slot is getting executed by N_i.
- **S4**: the task in the slot is being sent to node N_j i.e. it is being transferred from TB_i into IB_{ij}.
- **S5**: the slot holds a task and is currently not in any of the above states.

Task buffers can have multiple slots in states **S1**, **S2**, **S4** and **S5**, but for simplicity we will allow only one task at a time to be in state **S3**. We define the *buffer occupancy*, b_i of a node to be the number of slots in state **S5** at the current time. We say "TB_i is full" when the number of slots e_i in state **S1** is zero. In our model each node has a limited number of buffers.

```
OnRecvReqest(j, $b_j$) { // request from node j

        i = CurrentNode;
        $p_i$ = $b_i$ ; // pressure of node is equal to its buffer occupancy
        $p_j$ = $b_j$ ;

        if ($p_i$-1 > $p_j$) { // node has more tasks than requesting node
                $b_i$ = $b_i$ - 1;
                send(task, $N_j$); // send single task to $N_j$
                $e_i$ = $e_i$ + 1;
        } else {
                send(refuseMsg, $N_j$ ); // refuse $N_j$
        }
}
```

Figure 4.1: Protocol that nodes follow on receiving a request for a task.

A-FAST is divided into three sub-protocols for requesting tasks, responding to responses and performing a task. These algorithms have been provided in Figures 4.1, 4.2 and 4.3 respectively. The shaded portions of the protocols need to be performed in an atomic (synchronized) manner.

The advantage of this approach lies in the fact that the value of p is independent of the dynamic system parameters like P and C etc. Thus every node i can locally determine the value of its pressure p_i. The protocol gauges the system parameters by periodic querying instead, i.e. since each task transfer is followed by a subsequent request, barring the overheads of latency the protocol should make maximum use of the available bandwidth of a link. Similarly since each task completion is followed by another task, each node tries to make maximum use of its processing power.

Intuitively, the protocol should adapt to both a computation-dominated system as well as a communication-dominated one: faster nodes empty their buffers

```
OnRecvData(j, $r_j$) { // response from node j

    i = CurrentNode;

    if ($r_j$ is a task) {
        flag = true;
        while(flag) {
            if($e_i > 0$) { // there is an empty slot
                $e_i = e_i - 1$ ;
                transfer task from $IB_{ji}$ to $TB_i$ ;
                $b_i = b_i + 1$ ;
                requestData(j, $b_i$) ; // request more tasks from node j
                flag = false;
            } else {
                wait for a while;
        }
    } else {
        wait for a while
        requestData(j, $b_i$); // request tasks again
    }
}
```

Figure 4.2: Protocol nodes follow on receiving a response from a neighbor in return to a task request.

```
ProcessTask() {
    i = CurrentNode;

    if ($b_i > 0$) { // There exists some task
        dispatch task for processing;
        $b_i = b_i - 1$;
        perform task;
        $e_i = e_i + 1$;
    } else {
        Wait(till $p_i > 0$);
    }
}
```

Figure 4.3: Protocol that nodes follow to perform task computation.

faster and their *pressure* decreases, making them likely to receive more tasks. Similarly if a link is fast, tasks will be delivered faster across it, decreasing the pressure at the provider node, leading to more tasks being sent to that node. We will verify these claims in Section 4 through experimental verification of this variant of A-FAST.

4.2.2 Incorporating Reliability in A-FAST

In this section we show how our generic idea of pressure can capture other system properties such as reliability as well. To achieve this we shall modify the definition of pressure to incorporate fault tolerance into the scheduling strategy as well.

We define an unreliability parameter, τ_i, for each node in the system, which reflects the average time a node remains online. A fair estimate of the value of τ_i can be computed completely independently by each node. This can be done by maintaining a three tuple of <*num_of_readings*,τ_i,*last_val*> in the persistent storage of each node. Every time a node i comes online it increments the value of *num_of_readings*, sets τ_i to $\frac{(\tau_i + last\_val)}{num\_of\_readings}$, saves these values, assigns *last_val* to 0 and then begins functioning. The variable *last_val* is periodically updated to the elapsed time and saved back to persistent memory. The last recorded value of this variable can then be used as an estimate of how long the node remained online (the accuracy of this value will depend on the frequency of updates). τ_i is the average of all the values of *last_val* and gives an estimate of the expected duration node i is likely to remain online.

To incorporate reliability into A-FAST we modify our existing definition of pressure to be equal to $p_i = \frac{b_i}{\tau_i^K}$, where b_i is the buffer occupancy of node i and K is some real positive constant (we shall term it *Assurance Constant*) denoting the importance of fault tolerance to the system.

By doing this we make the pressure of a node inversely proportional to a power of to its chances of breaking down. Thus for two nodes with similar buffer occupancies, the node with a smaller value of τ (hence more unreliable) will have a higher pressure and therefore tasks will flow out of it towards a more "reliable" node. It must however be mentioned that giving too much importance to fault tolerance might have adverse effects on throughput, since slower but more reliable nodes will start getting more jobs assigned to them. This can be controlled by choosing an appropriate value of K. We will observe the effect of K more in the experimental evaluation section next.

4.3 Experimental Results

We wanted to test if A-FAST works for different types and scales of systems. We generated two types of networks topologies - internet-like graph topologies generated using the network-emulator package (NEM) [76] (G1) and cluster-like topologies (G2). For generating a graph with n nodes in G2 we built k clusters of equal size ($k \approx \sqrt{n}$). Nodes in these clusters were heavily connected (average connectivity of $k/2$). These clusters were then connected to each other in a random tree topology. The purpose of two totally different underlying topologies was to study if A-FAST's performance is topology dependent.

We generated graphs of four different sizes (n = 200, 400, 600 and 800). For each of the generated topologies, each node i in the graph was assigned a random processing speed, $P(i)$, uniformly distributed between 1 and MAX_SPEED. This value represented the number of tasks node i can process in one timestep. Every edge (i, j) was assigned a random value for $C(i, j)$, uniformly distributed between 1 and MAX_BANDWIDTH, denoting the number of tasks that can be sent along the link in a given time unit. For our simulations we set both MAX_SPEED and MAX_BANDWIDTH to the same value, namely 40. This was done to provide equal chances that the generated underlying system would be computation or communication dominated. All experiments were repeated multiple times (at least 5) and the average value of all the runs was taken.

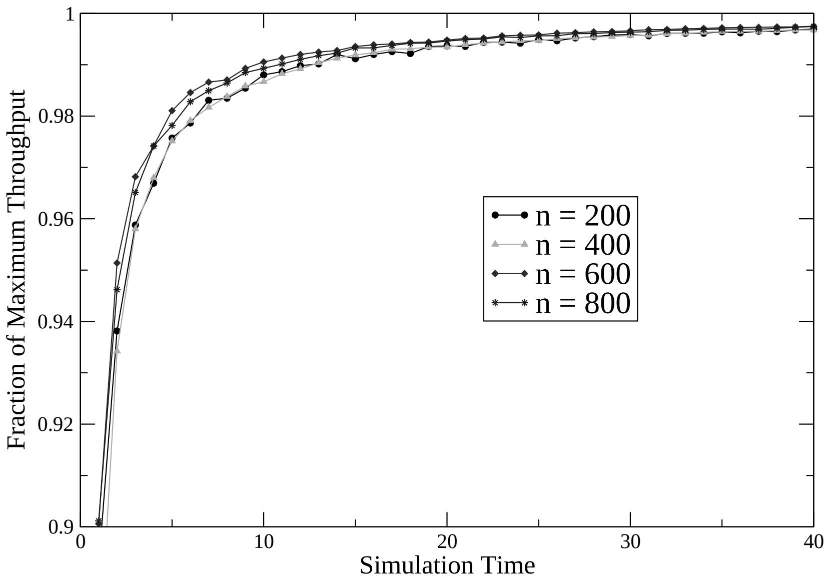


Figure 4.4: Performance of A-FAST in *Internet-like Graphs* of different sizes.

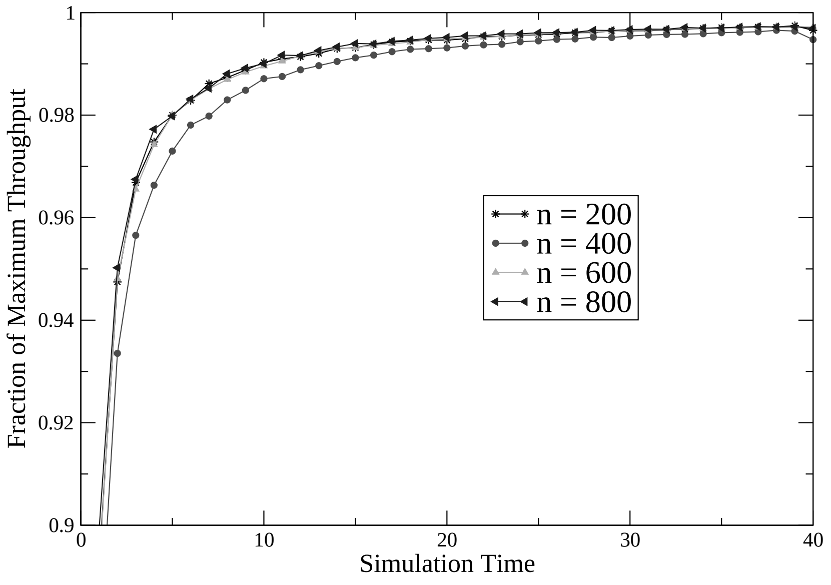


Figure 4.5: Performance of A-FAST in *Cluster-like Graphs* of different sizes.

4.3.1 Throughput

We compared the performance of the first variant (Section 4.2.1) of the protocol as a percentage of the maximum throughput of the system. The maximum throughput was calculated using the *maxflownet* package [67] on the generated network. The results for the two types of topologies, G1 and G2, are shown in Figures 4.4 and 4.5.

It can be observed that A-FAST performs very well, averaging over *99.5%* of the maximum throughput for both the topologies and all the different system sizes. This supports our contention that A-FAST is generic and scalable, although the size of our graphs are small. It can also be observed that the startup time of A-FAST is also small with almost all the simulations reaching 98% efficiency within 5 minutes of simulated time (5 minutes corresponded to approximately 750 completed tasks in our simulation setup)[2].

We also implemented a version of the RID algorithm [69] to compare its performance against A-FAST for communication-dominated systems. We generated these systems by generating the G1 type graphs described above but assigning link speeds that were less than the processing speed of the two nodes joining the links. The version of RID balanced the load every time the number of tasks in the Task Buffer fell below 5. The experiments were run for 60 simulated minutes to allow the RID algorithm to reach steady state throughput. The results are shown in Figure 4.6. We observe that while A-FAST achieves nearly 99.5% of the optimal throughput RID only achieves only around 95% of the optimal throughput. RID also takes a larger amount of time to reach the steady state throughput.

[2]It must be mentioned that we assumed zero latency networks for our simulations. This might be unreasonable in certain scenarios where the frequent exchange of request messages and the single task transfer approach of the protocol might affect the performance of the system. Section 4.4.1 shows how we can deal with scenario

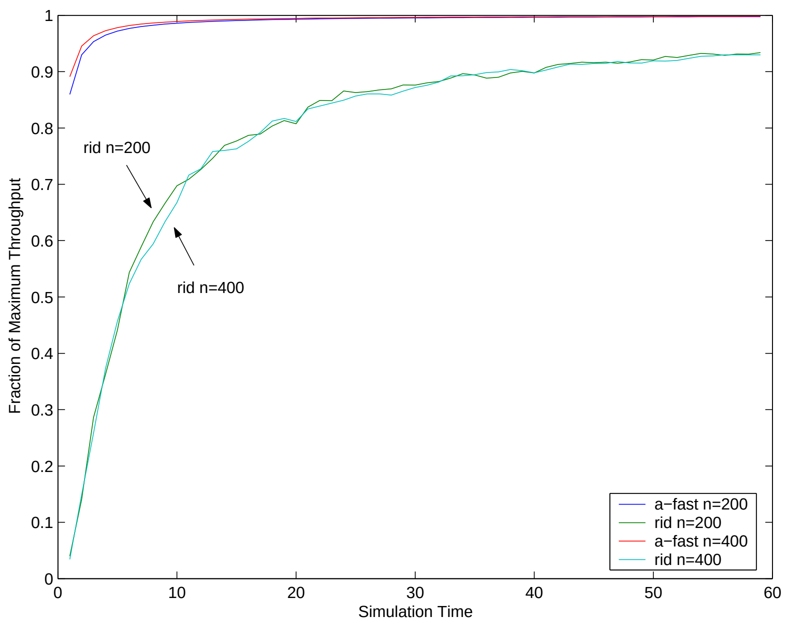


Figure 4.6: Relative Performance of A-FAST vs RID on *Communication-dominated Graphs.*

4.3.2 A-FAST in Dynamic Networks

For the experiments in the previous section the underlying system was static i.e. it did not change with time. Real systems over time can have their processor and link speeds changing. One of the strengths of A-FAST lies in the fact that it does not use the values of the processor speed, P, and the communication speeds, C, for its scheduling strategies. A-FAST's supply-on-demand approach coupled with the notion of pressure allows the protocol to adapt to system changes autonomously. To test this we repeated the experiments described in the previous section on G1 again, changing the underlying system (both node and link speeds) every 2 minutes of simulation time. Every node in the system had a 20% probability of changing by a magnitude of 20%. This meant that there was a fair chance of the system properties changing marginally every 2 minutes. The new value of maximum throughput was re-calculated after these changes. Different experiments were conducted to study the effect of increase and decrease of system performance.

The results are shown in Figures 4.7 and 4.8.

One can observe that A-FAST adjusts to system changes in an efficient and autonomous fashion. The yardstick used for comparison is the newly calculated *max-flow* value for the topology. While this value will always represent the optimal performance in systems with increase in performance A-FAST can do better than this yardstick (as it is evident from Figure 4.8) for systems with decreasing performance. This is because A-FAST can buffer up more tasks and then use them to sustain the performance temporarily even after the performance has decreased.

4.3.3 Reliability

In this section we test the validity of the Fault-tolerance-aware A-FAST variant described in Section 4.2.2. Each node in the system was randomly assigned a value τ uniformly distributed between $(5, 75)$. Since we conducted our experiments for 40 simulation time steps, this gave every node in the system an equal chance of failing or not failing in the life time of the experiments (a failed node did not restart). The nodes which failed were also uniformly distributed along the lifetime of the experiments. We then tested A-FAST with four different values of the unreliability constant K, denoting the importance of reliability for the experiments (note that for $K = 0$ the protocol reduces to the standard buffer based pressure approach described in Section 4.2.1 and is provided as a base case). We measured the change in throughput and amount of lost tasks (tasks that were assigned to nodes when they broke down). The results are given given Figures 4.9 and 4.10.

In all our experimental scenarios, the throughput of the fault-tolerant version of A-FAST achieves better throughput when compared to the standard version. Though the changes in throughput are not too large, it shows that the notion of *pressure* can successfully incorporate a range of features in it. However, we could not conclude anything definitive about the impact of K on throughput. This is because a smaller value of K reduces the importance of reliability and increases the chance of a potentially faulty node getting more tasks while a larger

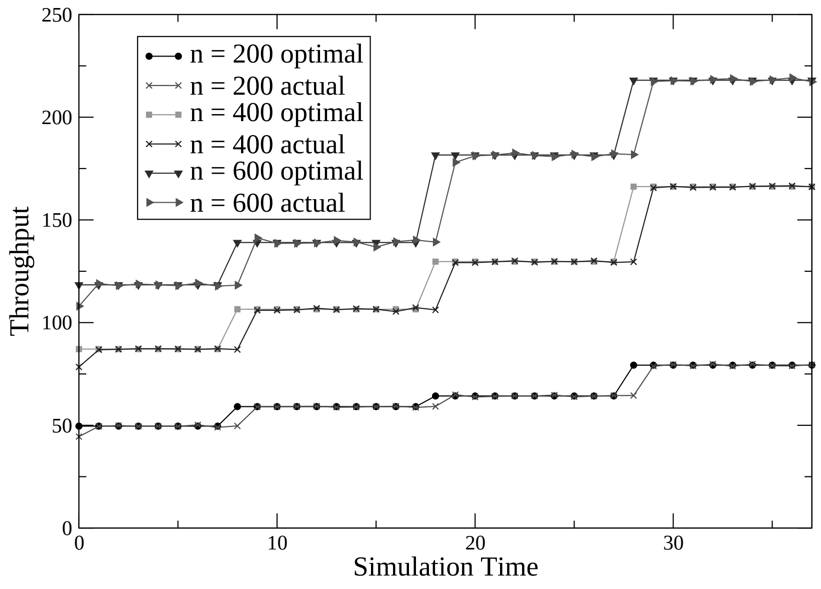


Figure 4.7: Performance of A-FAST in systems where system ability improves with time.

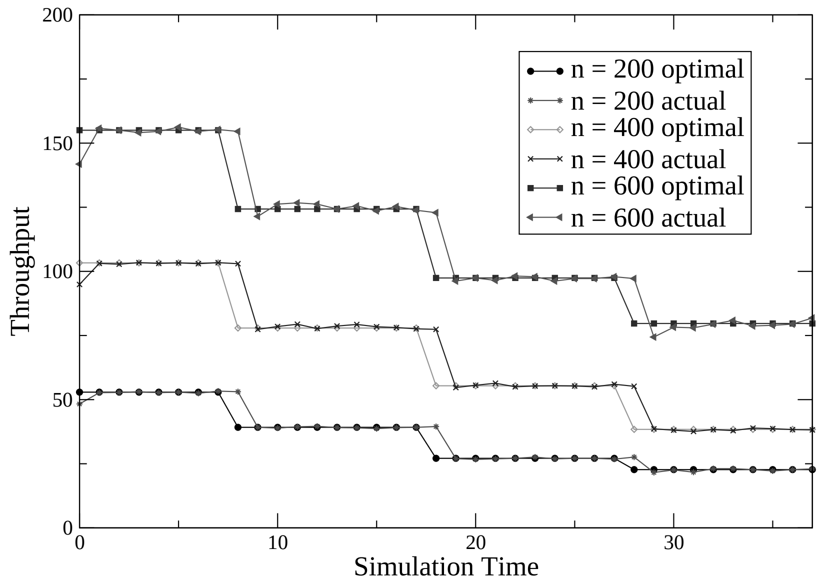


Figure 4.8: Performance of A-FAST in systems where system ability decreases with time.

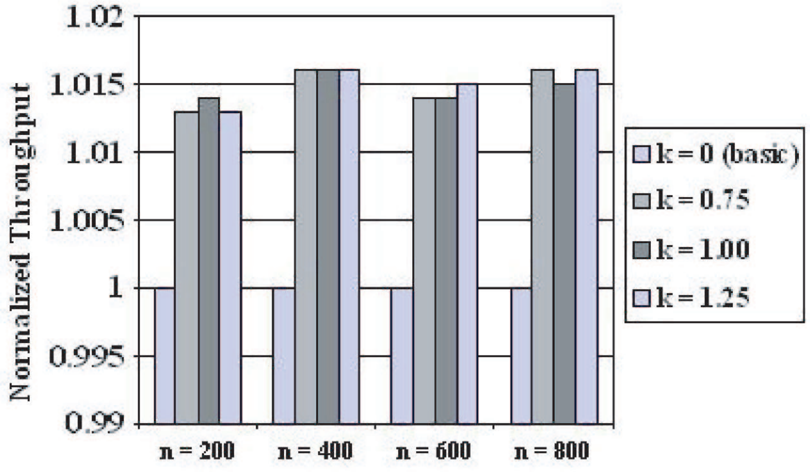


Figure 4.9: Effect of adding reliability to A-FAST on system throughput.

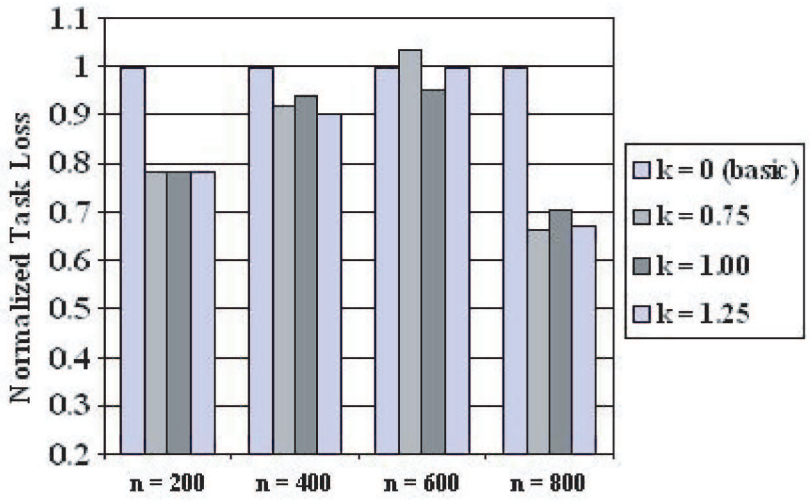


Figure 4.10: Effect of adding reliability to A-FAST on number of tasks lost (note: we do not deal with re-transferring lost tasks).

value of K might make slower and more reliable nodes get more tasks, thereby affecting performance. However, it is evident that the introduction of reliability as a parameter to pressure does pay off.

We, however, see a marked improvement in the reduction of task losses with A-FAST. In our simulations we did not take any measures when a task was lot. In a real system, a task loss might eventually require re-transmitting the task and reducing the task loss can eventually improve the system-throughput even further.

4.3.4 Practical Implementation

We also implemented the A-FAST protocol in a real system as part of our Java-based programming platform described in Chapter 5. This section describes some of the results and the lessons we learnt during the process. The experiments were run on the UCSD Fast Wired and Wireless Grid Project (FWGrid) [37] platform as a test bed. FWGrid is a cluster of Dual 1.6GHz AMD Opteron processors connected by high speed network. The system provides exclusive access to nodes i.e. a node is assigned to only one user at a time. We imposed a virtual topology on top of these clusters whereby nodes could only communicate with their immediate neighbors in the topology.

For our tasks we used an integration application where we tried to find the area under a curve by repeatedly adding up small rectangles under it. The task was divided into 160 equal sized sub-tasks (by dividing the X axis range of the integral uniformly). We tested the performance of A-FAST on three different topologies - star, a fully-connected graph and a binary tree. The results are shown in Figure 4.11. We have also provided an estimate of optimal performance by reporting the time $\lceil \frac{T}{N} \rceil$ as a yardstick where T is the time the task took to run on a single node (averaged over several runs) and N is the number of nodes used. Even though the results are preliminary in nature and system size was not too large, one can observe that the A-FAST's performance is comparable to the estimated

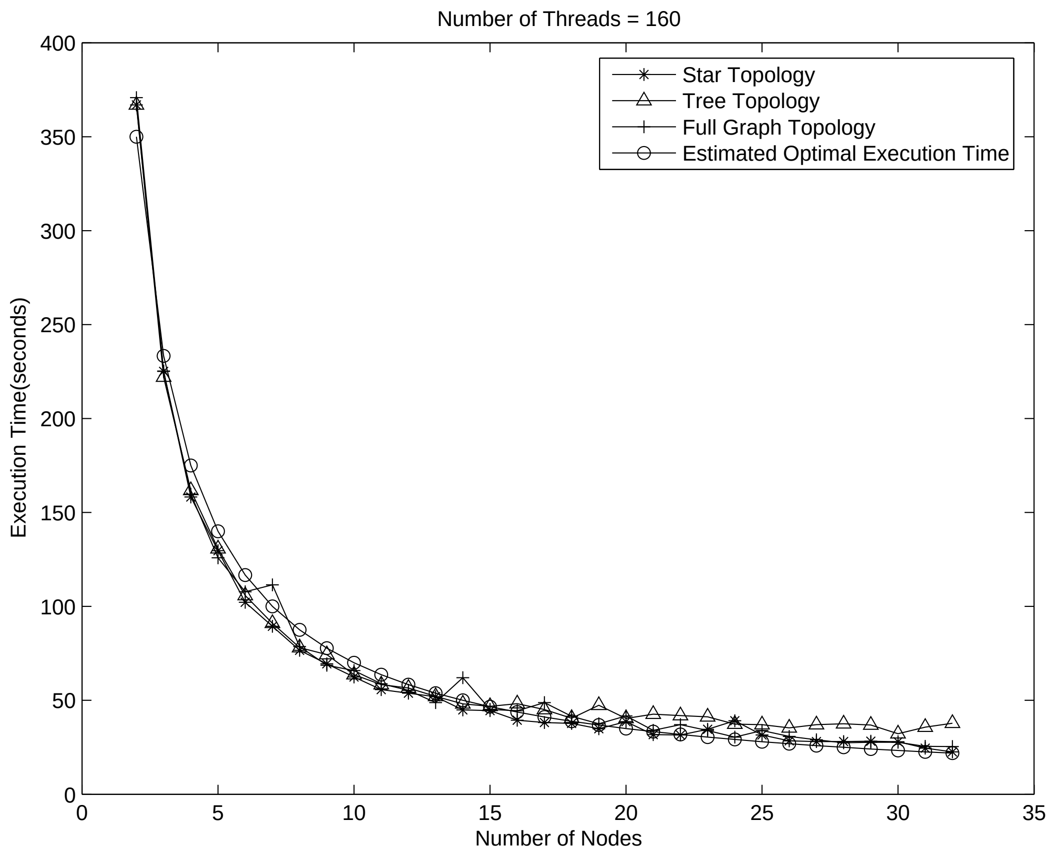


Figure 4.11: Performance of A-FAST in a real system with three different underlying topologies.

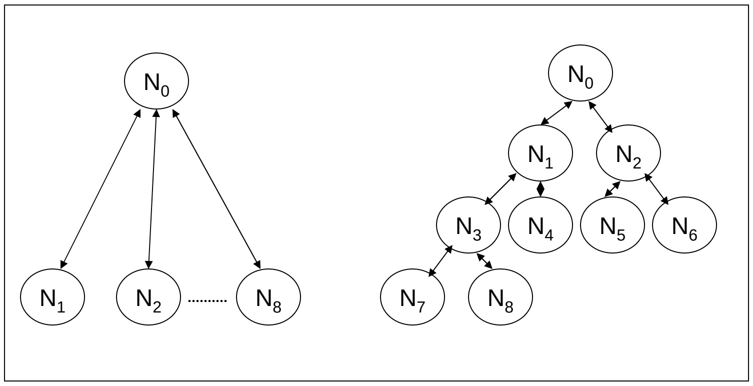


Figure 4.12: Example showing how topology can affect the maximum buffer capacity of a node. For the star topology shown above, having a large buffer capacity might allow one of the leaves to become the bottleneck by pulling more tasks than needed. For the tree topology however we want a higher buffer size for the non-leaf nodes (N_1, N_2, N_3) in order to facilitate better task transfer.

optimal throughput for all three topologies. A-FAST also scaled well over all three topologies across all the system sizes. This validates our claim that A-FAST is not only efficient as a scheduling technique but a practical one too.

4.3.5 Lessons Learnt

While implementing the practical version of A-FAST, described in the previous section, we ran into several interesting issues that are worth mentioning.

1. **Overlap of Computation and Communication**: Our earlier work [60] showed that even though it is traditionally assumed that computation and communication overlap perfectly, it is often not the case in practice. In an actual system (like our implementation) it is often the case that computation and communication get performed by parallel threads that are scheduled one at a time. By making unnecessary communication(requests) we might affect the performance of the computation thread significantly, thereby reducing the overall throughput of the node. In our experiments we experienced this effect

several times when excessive requests for tasks and unnecessary task transfer (to be explained in next section) affected the throughput of the node. It is therefore important to (i) avoid unnecessary communication and (ii) ensure that the individual threads performing computation and communication get scheduled proportionally (in our implementation, all threads were assigned equal priority).

2. **Buffer Sizes**: The basic A-FAST protocol described in Section 4.2.1 and our simulations assumed an unlimited number of tasks in the initial submitting node and also an unlimited buffer capacity in the participating nodes. However, for our implementation we were dealing with a fixed number of tasks. We were thus concerned about the overall running time and not the steady state throughput. While we didn't have to modify A-FAST much to adapt to this yardstick, we did run into some interesting issues due to our assumption of unlimited buffer sizes.

   Consider the two different cases - that of the star topology and that of the binary tree topology shown in Figure 4.12. In the star topology all participating nodes (except for the root) are leaf nodes. "Ideally" all the leaf nodes should get an equal number of tasks. However, in a real system the requests of nodes take place in multi-threaded fashion and it is likely that the root node gets more requests from one of the participant than the others. As an example consider a case when, due to thread management issues, the root gets more requests from N_1. Therefore N_1 ends up with more tasks than the rest of the nodes and becomes the bottleneck for overall runtime. Since all edges in our graphs are bidirectional in nature, it is expected that finally when N_1 is the only node left with tasks, the root will draw out tasks from N_1 and redistribute it to other nodes. However, this would be unnecessary communication and can affect performance further (due to the effect of communication on computation).

To solve this problem we put a limit on the buffer capacity of the nodes. Each node had a Maximum Buffer capacity (MB). Nodes checked if the number of outstanding tasks in the task buffer was less than MB before requesting further tasks. For the results shown in Figure 4.11 we set MB to 4 for the star and fully connected topologies. This ensured that no single node drew too many tasks without actually completing them. However, when we tried the same value of MB for the tree topology the performance reduced significantly. This happened because reducing the number of outstanding tasks in the task buffer made it more difficult for tasks to actually trickle down the system to other nodes. This happened because the possible pressure difference between nodes decreased when we set a low value for MB. This lead to a greater waiting period for nodes before they could get a task from their neighbors. With increase in the time to actually receive a task, the time to pass that task on to other nodes also increased (e.g. it would longer for the pressure in N_1 and N_3 to build and hence take even longer for tasks to reach N_7 and N_8. We thus noticed that the overall throughput of nodes decreased as we went down the tree. To deal with this situation we increased the value of MB for the tree topology. This improved our results considerably. For the results in Figure 4.11, the value of MB was set to 6 for the tree topology.

It must be mentioned though, that some of these issues only occurred because we did not have an unlimited supply of tasks and were concerned about the overall runtime and not the steady state throughput. However, since many applications fall into this category, it is important to understand the effect the underlying topology can have on the value of MB and how this affects the overall system throughput.

4.4 Other Issues

We now discuss some issues and special situations related to A-FAST and how they can be handled.

4.4.1 Latency

For all our simulations we ignored the effects of latency. This was done under the assumption that the size of a task and the time to transfer it will be significantly larger than the overheads of latency. Given that a lot of distributed computing tasks are long running and involve large volumes of data transfer, this is a reasonable assumption. However, if the tasks are small in size, with relatively small running times, then latency can have a noticeable effect on the performance, due to the per-task transfer approach of A-FAST. This can be dealt with by grouping a set of tasks together to increase the "effective task size". Even though this takes us away from the one-task approach of A-FAST that makes it adapt to both computation and communication dominated tasks, we will not lose out much since by assumption these tasks are small in size and runtime.

4.4.2 Avoiding Infinite Wait

With the suggested implementation of A-FAST it is possible for submitted tasks to get passed around the system without actually getting executed. One way to avoid this is to time-stamp the tasks. Since we assume that all tasks originate from a single node, this could be easily done without requiring system-wide synchronization. Participating nodes could just arrange the tasks in a queue sorted by their time-stamps and perform tasks from the front of this queue. This would ensure that earlier tasks get performed first.

4.4.3 Dealing with Multiple Submitting Nodes

Till now we assumed only one source (submitting) node. In an ideal autonomous system that supports collaborative community building, it is likely that multiple nodes will submit jobs into the system. We would thus want to ensure some kind of fairness among the various submitters.

One way to deal with this problem is to have a tag associated with each job, denoting its originating node. Jobs in a participating node can then be arranged in queues corresponding to their originating nodes[3]. Tasks should then be executed and sent to other nodes by traversing these different queues in a circular fashion. This ensures that all users who have submitted jobs get equal priority in terms of both communication and computation[4].

It must also be mentioned that the time-stamped approach described in the previous section to avoid infinite wait applies to this setting as well. Each submitting node can time-stamp its jobs locally without worrying about the system-wide nature of the time stamps (hence we do not need system-wide synchronization of the relative values of the time-stamps). Jobs will then get selected, performed and passed on to other nodes in a circular fashion giving equal priority to all submitting nodes. Thus, as long as each node time-stamps the jobs it submits, the jobs will get a priority equal to all other jobs submitted by other nodes. Since jobs will also get prioritized based on their time-stamps (even though within a single node) it will also prevent them from getting into infinite waits.

In this chapter we studied the A-FAST protocol for autonomous scheduling of homogeneous independent tasks. A-FAST achieves $\approx 98\%$ of the optimal steady state throughput and scales well to different systems sizes and changes. We also discussed variants of A-FAST that can incorporate greater reliability and support multiple submitting nodes, without compromising on the autonomous nature

[3]It is not necessary that there is one unique queue corresponding to each originating node since that would affect scalability. In case there are too many originating nodes then jobs can be hashed to queues based on their originating node.

[4]It must be mentioned that we still assume all jobs to be of similar nature. If different users submit different jobs the problem scenario changes completely.

of the protocol.

Portions of the text of this chapter are a reprint of the material as it appears in [71]. The dissertation author was the primary researcher and author and the co-authors listed on this publication directed and supervised the research which forms the basis of this chapter.

Chapter 5

Programmability

One of the biggest challenges towards making distributed systems more popular is to facilitate the development and deployment of a wide range of applications for them. The various complexities associated with these systems make it difficult for programmers to develop code that makes efficient use of available resources. Autonomy and heterogeneity makes the problem of code development for these systems even more difficult by removing centralized sources of information that developers can use. In this chapter we propose a system architecture that allows developers to produce code for these systems by giving them a simplified view of the available resources. We then show how the scheduler can make runtime decisions to adapt this code for various scenarios, without requiring the programmer to change his code.

5.1 Introduction

Decentralized control, coupled with the growing heterogeneity of individual nodes of large scale autonomous computing systems, makes programming extremely challenging. Code that is fine-tuned to perform efficiently for a given set of resources may not scale to a larger or different set of available resources. On the other hand, code that is generic runs the risk of of substantial overheads that lead to sub-optimal performance. In the presence of centralized monitoring and control, programmers can gather detailed information about the system (size, topology, processing strength etc.). They can then use this information to produce optimized code that makes efficient use of the resources. In a dynamic and autonomous computing environment programmers have a very limited view of the system and its abilities. It is likely that, in a truly large-scale autonomous computing system, resources will be continuously added and removed from the system. The system topology, size and ability is likely to change too. It will be extremely difficult for the programmer to write code for such a dynamic system without having complete knowledge about it. This is the problem that we try to address in this chapter.

We propose a system architecture that provides the programmer with a virtual view of the system as a given number of *virtual homogeneous resources*. This view makes it easier for programmers to produce efficient code by hiding several complexities of the underlying system. Nevertheless, as our experiments show, this *virtual view* of the system allows the code to adapt to the parameters of the underlying system, without having to change the original program or involving the programmer in the decision making.

To motivate our approach, consider two common programming options for distributed systems - MPI or PVM-like approach [70, 99] and an ATLAS [8, 14] like programming model. In the former case, the user normally deals with a pool of homogeneous nodes. The programmer breaks up the given task into a fixed number (usually equal to the number of available resources) of sub-tasks that are then assigned to the various resources. This allows the programmer to fully specify the granularity of parallelism and easily address issues such as message passing and scheduling. The systems for which PVM was initially designed were "client-server-like" by nature, where the server assigned tasks to individual client resources.

In a decentralized system where nodes only interact with their immediate neighbors, it is possible that the local view of the system is very different

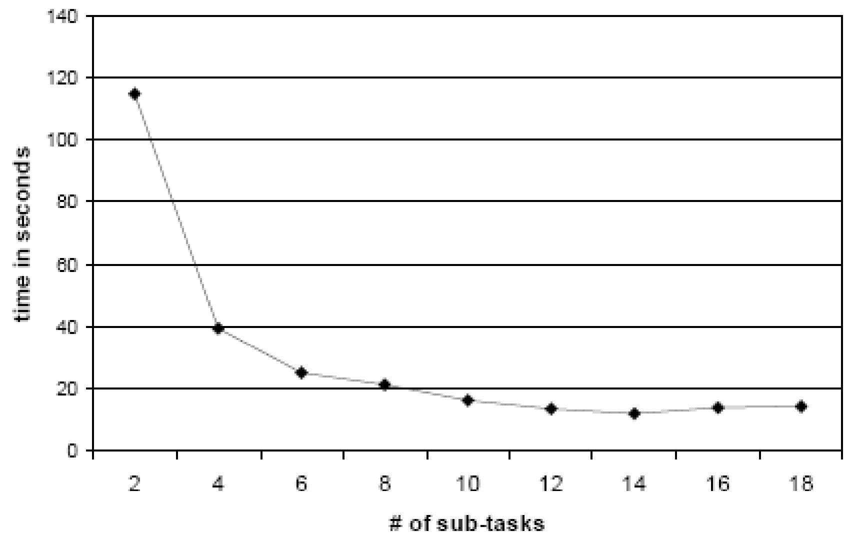


Figure 5.1: Effect of number of subtasks on performance across two nodes of dissimilar capabilities.

from its overall capability. This suggest that provisions for allowing the tasks to get progressively divided into smaller sub-tasks, based on the run-time ability and topology of the system, are needed. Moreover, given that many present day distributed systems consist of heterogeneous nodes with varying capabilities, dividing a task into as many subtasks as resources might lead to performance degradation. Figure 5.1 illustrates this possibility, where a divisible task was executed across two nodes, a Pentium IV 1.5GHz PC with 512MB of memory and a Pentium III 600 MHz machine with 256MB memory. Creating just two tasks makes the slower node the bottleneck of the system and affects performance significantly. This suggests the need for an easy interface for feeding such runtime information into the program.

On the other end of the spectrum, in ATLAS (and similar systems like Satin [100]) the programmer is expected to progressively break up tasks into smaller sub-tasks that are then spread across the system using Cycle Stealing [14]. This task division is done by the programmer and does not necessarily depend on the configuration of the underlying system. Breaking up a divisible task progressively

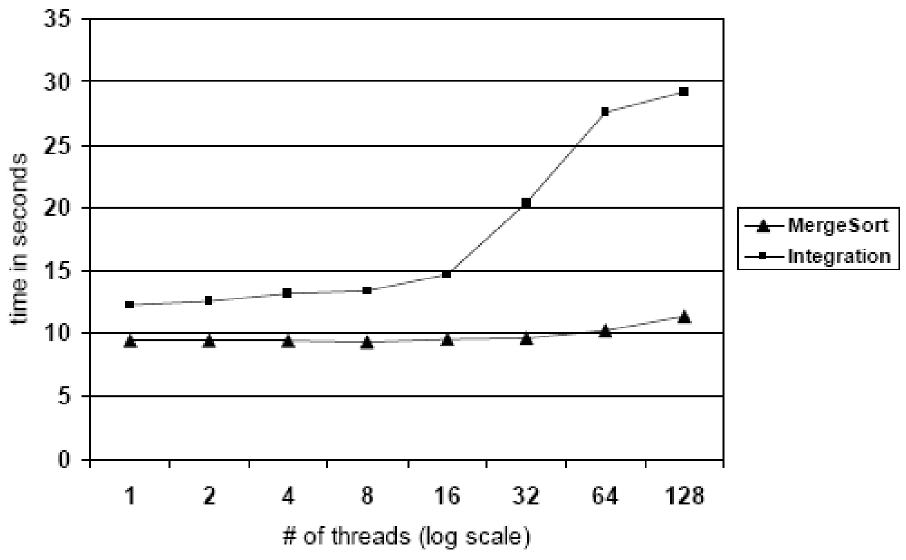


Figure 5.2: Effect of increasing the number of threads on performance of a single node. While the performance degradation for MergeSort is smaller, Integration gets affected more severely.

in the absence of adequate resources can be detrimental to the performance of the program, creating unnecessarily small threads that can adversely affect performance. Figure 5.2 illustrates this point. We plot the performance of two simple programs (a merge sort and an integration function that adds up small rectangles) run on a single machine (Pentium IV, 1.5 GHz, 512MB), by varying the number of threads. While the performance degradation is much smaller for Merge Sort (because at any given instant at least half the threads are waiting for the other threads, and the memory and computation operations can also be overlapped), there is a significant drop in the performance for the computation-dominated integration task. Thus increasing the number of threads without having adequate resources can affect code performance adversely. Creating more tasks than needed can also increase the communication overhead significantly. Thus while an ATLAS like approach can work very well for a large set of tiered resources in a computation-dominated environment, it might not work as well in other scenarios.

We propose a programming model that lies between these two - where the programmer produces divisible code for a number of *virtual resources* but this number is given to the program at runtime through a node's scheduler. The actual granularity of a job is thus decided at runtime. The ATLAS like divisible structure allows the program to be divided into smaller tasks if additional resources are available, while the runtime information provided by the scheduler ensures that the granularity is in proportion to the actual scale and heterogeneity of the system, and doesn't affect performance adversely. We also show how using this simple programming model allows the scheduler to adapt to several runtime scenarios. In summary, the main contributions of this section are as follows:

- A simple programming model based on a virtual resource interface that eases the programmer's task by hiding the complexities of the decentralized platform.
- A minimal API through which the program and the decentralized system interact at runtime.
- A Java-based systems architecture which supports adaptive parallelism, and different scheduling and communication strategies, without involving the programmer.
- Experiments using FWGrid which show how the use of this architecture can lead to significant performance gains in several different decentralized runtime scenarios, including different numbers of nodes and network topologies, for both compute- and communication intensive applications, without having to change the program or involve the application programmer.

The rest of this chapter is organized as follows. We begin by discussing the architecture of our proposed system and shows how code produced can scale using different network topologies and sizes and then present results from our experiments on the FWGrid system.

5.2 System Architecture

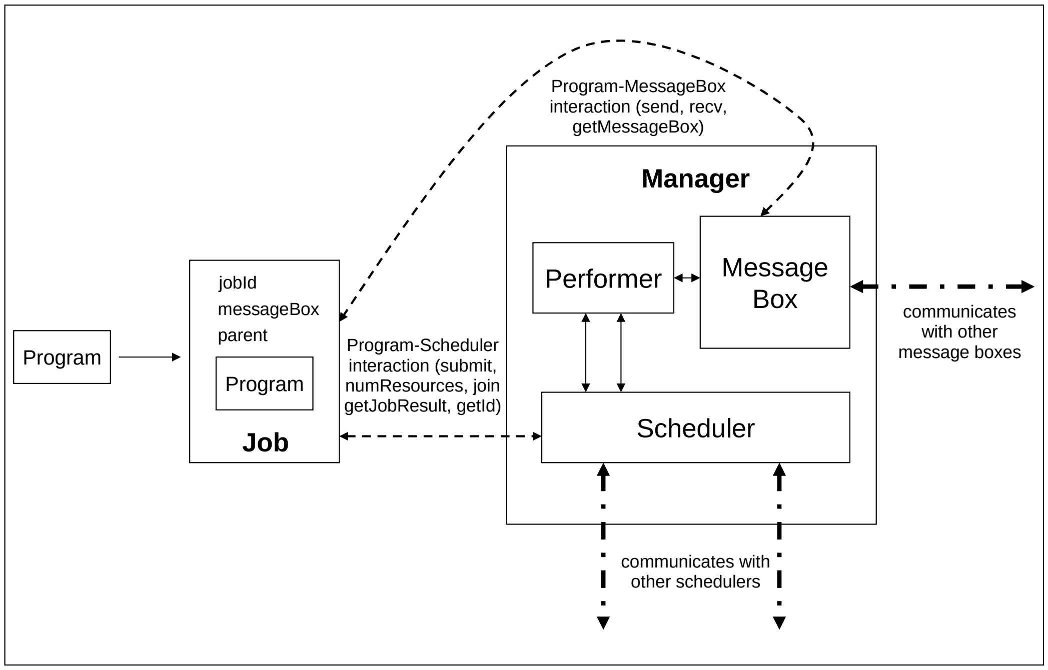

Program-MessageBox interaction (send, recv, getMessageBox)
Manager
Performer
Message Box
communicates with other message boxes
jobId
messageBox
parent
Program
Program
Job
Program-Scheduler interaction (submit, numResources, join getJobResult, getId)
Scheduler
communicates with other schedulers

Figure 5.3: Overall system architecture of our framework. Each node has one Manager and a job it interacts with. Managers of neighboring nodes interact with each other.

In this section we describe the architecture of our proposed system. We present the interface between the program and the scheduler, the communication paradigm used, and the components of the architecture. Figure 5.3 gives a basic overview of our system. The system has two principal components:

- **Job**: A Job in our system is any Java program that extends the Thread class. All task transfers and execution in our system takes place at the granularity of a Job. We encapsulate the program within a Job class that automatically embeds additional information, such as a unique *jobId* and the associated *Message Box Id* (used for message passing), in the code. Programmers can thus create a stand alone Java task that can execute as a thread and encapsulate it as a Job and make it executable in our system. Jobs also have a *groupId* and *priority* field that can be used to group them together and assign relative priorities to them. We will later discuss in Section 5.2.2 how these two fields can be used to deal with situations like irregular tasks.

- **Manager**: Each node in our system has a single Manager application running on it. The Manager is a collection of utilities that we felt should be independent of the program. The programmer can therefore develop code that is independent of underlying system issues and allow the Manager to deal with them at runtime. In our system a Manager consists of a Scheduler (that schedules the various submitted jobs and interacts with Schedulers of neighboring nodes), a Message Box (which acts as a message repository that the jobs can use to communicate) and a Performer (the component that actually runs a job and deals with local Thread Management).

By keeping these two components separate in terms of functionalities we ensure that the programmer doesn't have to worry about system-specific issues. The Manager provides the programmer with a simplified virtual view of the resources available that hides many of the underlying complexities that are dealt with by the Manager. The programmer has the application-specific knowledge to break up the program into subtasks; the scheduler has the knowledge and responsibility to adapt to runtime conditions. All interactions between a Job and the Manager happen through a well-defined API. It is therefore possible to replace an existing scheduler with a new one that might be better suited for a specific environment, while the program remains oblivious to the change. This is supported by our experience: during our experiments, we changed the scheduling component several times to meet particular system specific needs, but these did not require any changes to the original program. We next describe the API through which the program and the Manager interact.

5.2.1 The Interaction API

Our objective while designing the system was to choose a simple and minimal interface between the program and the Manager to ease the programmer's effort, yet rich enough to allow code to run efficiently in different scenarios without additional programming effort. The following are the basic methods that our system provides for a Job to interact with the Manager.

- *void submitJob (Job j)* - Used by the program to create and submit Jobs to a virtual resource's Scheduler. Any job in the system can thus create further (sub)jobs and submit them to the system and facilitates adaptive parallelism. Note that the jobs created need not be aware of the location where they get created or are submitted.

- *int numResources ()* - Used by the program to query a node's Scheduler to find out how many virtual resources are available. This is one of the biggest features of our system where the Manager provides the program with a virtual view of the size of the system. The programmer is expected to believe that the returned value represents a pool of homogeneous nodes that are similar in their capabilities. This simplifies the programmer's view of the system. We will later show how, by varying the value of this number, the Scheduler can adapt to a range of situations at run-time, ensuring scalability, efficiency and fairness.

- *void join (Id jobId)* - Used by a program to synchronize Jobs where Jobs can wait for a particular Job (identified by the jobId) to complete before continuing execution.

- *Job getJobResult (Id jobId)* - Used to retrieve the results of a completed Job. If a Job is incomplete at the time of invocation the method waits till the result is available and then returns it. Programmers are thus expected to gather their results as part of the Thread Object they create and encapsulate with the Job class. They can then retrieve the associated Job and extract the actual result from the Object. Another way of retrieving result(s) would be to use the message passing Section 5.2.3. The example given in Figure 5.4 will explain this further.

- *void send (Id from, Id to, int messageId, Object message)* - Used to send a particular message to a desired Job (identified by its unique id).

- *void recv (Id from, Id to, int messageId, Object message)* - Used to receive a particular message from a desired Job (identified by its unique id).

Other than these methods that the Manager must provide for the program(s) to function properly, the manager must also provide two other methods that the Job class uses to hide necessary information from the programmer. These are:

- *Id getId ()* - This method is used to assign each Job a unique system-wide jobId. Every time the programmer creates a new Job this method is automatically invoked to assign a unique id to the Job.

- *MessageBox getMessageBox (Id id)* - A Message Box in our system is like a mailbox where messages for a Job are stored. This method is used to assign a Message Box to every Job that gets created. This message box is then used to send all messages intended for that particular job. We later show in Section 5.2.3 how this method can be used to prevent any one node from becoming a data bottleneck in the system.

Having presented the basic API that the system provides, we now look at an example program to better understand the various issues. Figure 5.4 shows a sample program written using our API. The program finds the integral of a function between two limits. When the program starts execution on a node, it queries the number of locally known, virtual resources, and assumes all resources are similar

```
int numOfNodes = Manager.numResources(); // gets the number of nodes
if(numOfNodes == 1)
{
            result = doIntegration(); // just perform the integration by adding the rectangles
}
else
{
            Job rThreads[] = new Job[numOfNodes]; // create as many Jobs
            double interval = (end-start)/numOfNodes;

            for(int i=0; i<numOfNodes; i++) // assign each Job a range
            {
                        rThreads[i] = new Job(new Integrate(start+i*interval,start+(i+1)*interval,range));
                        Manager.submitJob(rThreads[i]);
            }

            for(int i=0; i<numOfNodes; i++) //  wait for each job to finish and combine results
            {
                        Manager.join(rThreads[i].jobId));
                        result += ((Integrate)((Job)Manager.getResult(rThreads[i].jobId)).job).result;
            }
}
```

Figure 5.4: Example program (calculates the integral of a function between two points) to perform integration in our system. The program adaptively divides itself at runtime to divide itself according to the number of available virtual resources. The calls made to the Manager using our proposed interface is shown in bold.

as part of the programming methodology. The program is written to break up the task, based on this number, by the programmer, who can use application-specific knowledge. These subtasks can then be executed on the original node, or passed on to any other nodes in the system, as determined by the node's Scheduler. In turn, when a subtask executes a job on this new node, it queries the new node's Scheduler. If the Scheduler says there are additional resources, then the job will be further broken down adaptively to make use of these additional resources. Thus, even though all decisions are made locally, the program adapts to the actual runtime environment.

We will later see how the Scheduler can tweak the value of *numResources* to adapt to various situations. It should also be mentioned that the value of *num-Resources* and the actual breaking down of a job into sub-jobs is done at run-time. Thus the program can behave differently when run under different environments without the programmer being aware of it.

We now provide details of our implementation of the Scheduler, Message Box and the Performer. Once again, it must be mentioned that the system programmer is free to provide any implementation of the these three components and the program will still run as long as the Interaction API remains constant.

5.2.2 The Scheduler

The Scheduler (S) is the component of our system that is largely responsible for deciding the granularity at which tasks are broken down and also for scheduling these tasks on other virtual resources. The program queries the Scheduler to find the number of available resources in the system at runtime, based on which the actual sub-task creation takes place. The interesting aspect of this interaction is that the Scheduler can report the number of resources to the program to be other than the exact number in the system, and the program will still be run successfully to completion. The Scheduler can therefore set this reported number to meet several requirements, without involving the programmer. As an example consider the following two situations where "misreporting" the value of *numResources* to the programmer actually aids the process of scheduling.

1. *Scenario 1* - Suppose all the neighbors of a node are currently busy performing some task. In such a situation the Scheduler should ideally report the number of resources to be 1 to the program. In this case, the program will then not break itself into smaller tasks, and just run as is to completion. However, if at a later time, any neighbor of the node frees up, then there is no way of distributing the work dynamically to it[1]. Instead the Scheduler can report a higher value to the program (say 3), so the program will break itself up into three tasks. In the worst case scenario the node performing the job does all three sub-jobs itself (which will still be a smaller overhead compared to a static partition approach taken by ATLAS, as illustrated in Figure 5.5 in Section 5.3.1). However, if a neighbor frees up later, the Scheduler can assign the sub-job to it to speed up the execution.

2. *Scenario 2* - This is the same scenario as shown in Figure 5.1, where there are two neighbors available to a node, and the Scheduler reports back the number of nodes as 2 to the program, which then divides itself into two sub-jobs that are then assigned to each node. However, if one of these nodes is twice as fast as the other, then the slower node will become the bottleneck. The Scheduler could have instead reported the value 3 to the program, resulting in 3 sub-jobs which can be divided between the neighbors in proportion to their processing speeds.

Hence we see that it might actually be beneficial to either increase or decrease the reported value of *numResources* to the program. While in our programming methodology, the programmer assumes the reported number of homogeneous resources, the Scheduler can take care of issues such as heterogeneity, fairness and dynamic property of the system by setting this number accordingly.

[1]The programmer can insert regular checks in his code but that makes the job of the programmer difficult and fails to serve the purpose of this work.

The Scheduler is also responsible for propagating jobs by interacting with its neighboring Managers. In our current implementation, we use a modified version of the A-FAST protocol described in Chapter 4 to schedule across nodes. Each node runs a Manager, and the Scheduler of neighboring Managers query each other periodically. Jobs are passed along nodes one at a time from a region of higher pressure to lower pressure, where *pressure is defined as the number of outstanding jobs + number of running jobs*. We modified A-FAST by putting a bound on the maximum size of the task buffers. A-FAST allows jobs to trickle through the system, while avoiding cycles (thus unlike ATLAS we can deal with arbitrary network topology, not just trees) and maintaining efficiency. While reporting the value of *numResources* to a job, our A-FAST Scheduler starts by reporting a value that is proportional to the number of neighbors, and thereafter adapts itself by reporting a value that is a multiplicative factor of the sum of the *pressure differences* between the node and its neighbors[2].

In Section 5.2 we mentioned that the programmer can group jobs together using their *groupId* field and prioritize them using their *priority* field. The programmer can give a group of jobs the same group id[3]. Jobs can also be prioritized in a relative scale ranging between 0 and 1. This priority is used to *hint* the Scheduler about the relative importance of jobs within the same group. It must be noted that this is an added feature for the programmer (in case he wishes to prioritize his tasks or implement more sophisticated techniques such as irregular partitioning) and can be completely ignored if the programmer considers all tasks to be uniform in nature. It is the responsibility of the Scheduler to *try* and assign the higher priority tasks of a group to the faster available resources. Appendix B describes some of our ongoing work on how such a priority can be set effectively by users and how the Scheduler can use this value for scheduling a pool of tasks

[2]One can consider propagating information along the system to get a better estimate of the actual number of nodes but such techniques are beyond the scope of this paper.

[3]In our present implementation we do not provide a mechanism to ensure that jobs provided by different users do not have conflicting group ids but this is not very difficult to enforce using the Manager to generate group ids.

efficiently.

Once again, it must be mentioned that the program is created independently of the scheduling algorithm used, and one can replace A-FAST with any other scheduling technique without affecting the functioning of the program.

5.2.3 The Message Box

We now describe the communication mechanism supported by our system. As described earlier, all communication takes place using just two methods - *send* and *receive*. In our current implementation both these calls are blocking in nature.

The basic abstraction for message passing in our system comes in the form of a Message Box. Each node in our system has a single Message Box where it saves all incoming messages. Messages are saved by hashing them to a key value using their *from*, *to* and *id* fields, which uniquely identifies every message in the system. Whenever a Job is created it gets a Message Box assigned to it. All messages directed to the Job then gets sent to (and retrieved from) the assigned Message Box.

As mentioned in Section 5.2.1, every Manager provides a *getMessageBox* method which allows a node to locate the Message Box associated with a job. When a Job gets created it invokes the *getMessageBox* method with its unique *jobId*. Each Manager keeps an index of all jobs created on it along with their associated Message Box locations. When the Manager realizes from the Id of a job that it has never been assigned an Id (implying that the job is getting created) it assigns it a new Message Box. It must be mentioned that it is not necessary that the assigned Message Box be on the same node as the one where the Job is created. This allows nodes to share the message passing load if they desire, without involving the programmer in the process. This mechanism also allows messages for a Job to be moved around to a new Message Box as long as the parent node of the Job makes a record of it.

Whenever a *send* or *recv* call in made, the local Manager invokes the *getMessageBox* method of the Job's parent node (which is embedded in the Job) and identifies the Message Box from where the actual data has to be retrieved. If the node realizes that Message Box location is the same as itself, it just returns (or saves) the desired value. Otherwise the request is passed on to the destination Message Box. The entire process is thus transparent to the program or the programmer and prevents any one node from becoming the data bottleneck in the system. Moreover, in a decentralized environment it is possible that tasks travel across the system to many nodes before finally getting executed; our technique and architecture can prevent unnecessary data transfer by shipping data only when a job needs it.

In our implementation when a Job gets created it is assigned the Message Box located in the node where it is created. If the Job creates further sub-jobs, then they all get assigned the Message Box of the node where the sub-jobs are created. Thus if a Job J_1 is created in node N_1 and creates further sub-jobs J_2 and J_3 then both these Jobs get assigned the Message Box in N_1. However, if J_2 is finally assigned to node N_2 and creates new sub-jobs J_4, J_5 and J_6, then all three new sub-jobs get assigned to the Message Box in N_2.

It is also possible to implement fancier message handling (e.g. buffering and prefetch information from a remote mailbox) and message box assigning options that improve performance. Our architecture makes it possible to try any of these options without changing the original program as long as the *send*, *recv* and *getMessageBox* methods are implemented correctly.

5.2.4 The Performer

In our system the Scheduler interacts with other managers to delegates Jobs, but the actual Jobs must still be executed. Since resources that the Scheduler reports available via *numResources* might indeed might not be there, it is the system's responsibility to ensure that Jobs get completed in their absence. Running every created Job as a parallel thread may adversely affect performance, and launching only one thread at a time might prevent a Job from completing. This makes *thread management* a very important issue. The Performer both schedules and executes threads at the node. In our current implementation, each node launches a maximum of 2 Jobs simultaneously (since FWGrid consists of dual-core processors). Every time a Job finishes, the task queue is checked, and if there is a waiting Job, it is scheduled. However, it is possible that two threads are not enough to complete a Job. Our present implementation of the Performer periodically searches for blocked Jobs (via the Manager, which can easily detect them since all blocking methods are invoked via the Manager) and launches the necessary additional Jobs to eliminate blocking.

5.3 Experimental Evaluation

In this section we present results from experiments where our objectives were twofold - (i) to generate parallel code for different types of applications (both computation and communication based) and (ii) run these programs under different scenarios (scale, topology etc.) to show their performance without having to change the original program.

We used the UCSD Fast Wired and Wireless Grid Project (FWGrid) [37] platform as a test bed. FWGrid is a cluster of Dual 1.6GHz AMD Opteron processors connected by high speed network. The system provides exclusive access to nodes i.e. a node is assigned to only one user at a time. We imposed a virtual topology on top of these clusters whereby nodes could only communicate with their immediate neighbors in the topology.

We now describe the various experiments we conducted and the issues we faced in adapting to different situations.

5.3.1 Computation based tasks

The results of the practical implementation of A-FAST shown in Figure 4.11 were performed using the architecture described in this chapter. As mentioned in Section 4.3.5, we faced an interesting situation while dealing with the three different topologies. A-FAST's pull-based model led to situations with the star and fully connected topologies where a single node would draw more tasks than needed, thereby starving the other nodes. This made the single node with more tasks the bottleneck of performance. To combat this problem we put a limitation on the maximum number of outstanding tasks that a node could have, which worked well. However, the same approach, when used on the tree topology, led to significant performance degradation. We realized that this was because of the small bound on buffer size in the Manager prevented tasks from spreading across the system. Consequently, for the tree topology, we increased the buffer size and this improved the performance significantly. Thus we used two different scheduling strategies (one for the tree topology and another for the remaining topologies) but we did not have to make any changes to original program. The clear separation of the Manager and the Program allowed us to replace a scheduler that is better suited for a scenario without having to change the original program at all. This is precisely what we had hoped to achieve.

We then tested the effect of varying the reported value of *numResources* on the performance. Figure 5.5 shows these results for the tree shaped graph. Nodes reported a value of *numResources* that was a multiplicative factor of their number of neighbors. We see that increases in this factor improved performance up till a certain level (even better than a constant value of 160), after which the performance degraded. This shows that the granularity at which a job is broken down is important and can affect performance significantly (hence an ATLAS-like repeated breaking down of jobs in not advisable). It must be noted that we have not determined the optimal granularity, but our system provides an architecture where the programmer can be freed from the responsibility of dealing with this

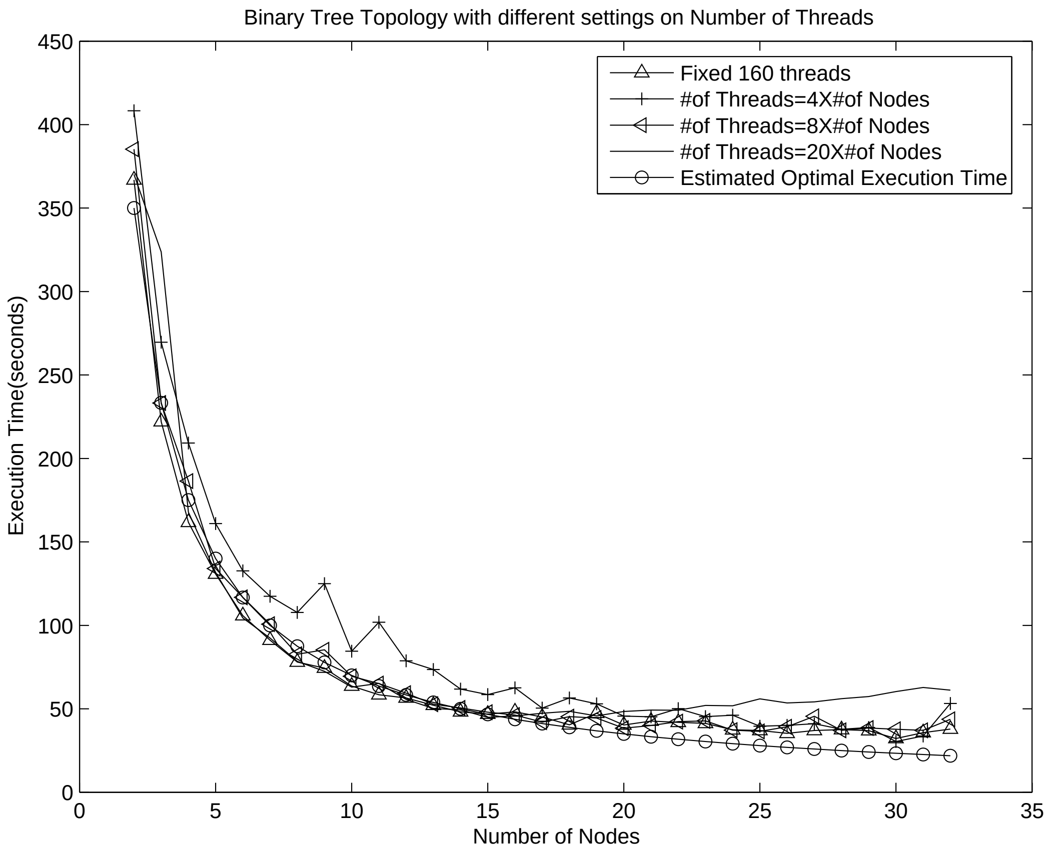


Figure 5.5: Effect of the value of *numResources* (equals to the number of threads created) on performance. It can be observed that increasing the reported value of *numResources* improves performance till a certain point, beyond which the performance degrades. The Scheduler can thus decide on the granularity of task division at runtime based on the underlying system, without involving the programmer in the process.

issue and the scheduler can decide the granularity at runtime.

5.3.2 Communication based tasks

To test how our architecture handles tasks with significant communication, we use a 4-point stencil Jacobi Iteration method as our sample program. The initial array size was around 1GB. The job was broken down into further sub-jobs that were then transferred around the system. Each sub-job got assigned an equal set of consecutive columns of the array. Before each array iteration began, jobs dealing with adjacent columns exchanged border information (through message passing with their neighbors). This example also illustrates synchronization between jobs (an iteration can only begin when the previous iteration of the neighbors of a job are over). We used the message passing technique described in Section 2.3. We tried two different strategies for assigning Message Boxes to the jobs:

- **Local**: where the Message Box of the node where the job was initially submitted was used as the default Message Box
- **Remote**: where all nodes interacted with a separate remote Message Box running in a node that did not run any tasks (like a Data Server).

While the former strategy has the advantage of making many of the data accesses local to the node (hence avoiding expensive remote invocations) it also runs the risk of interfering with the actual task running ability of the node, slowing down performance. The results are shown in Figure 5.6 for the star and full graph topologies.

We see that all the experiments scale reasonably well till a certain problem size. We can also see that for the local experiments, where the Message Box was on one of the running nodes, the performance is better for smaller system sizes since most of the data accesses could be made locally. Our architecture made it possible to try out two different data handling techniques without having to change

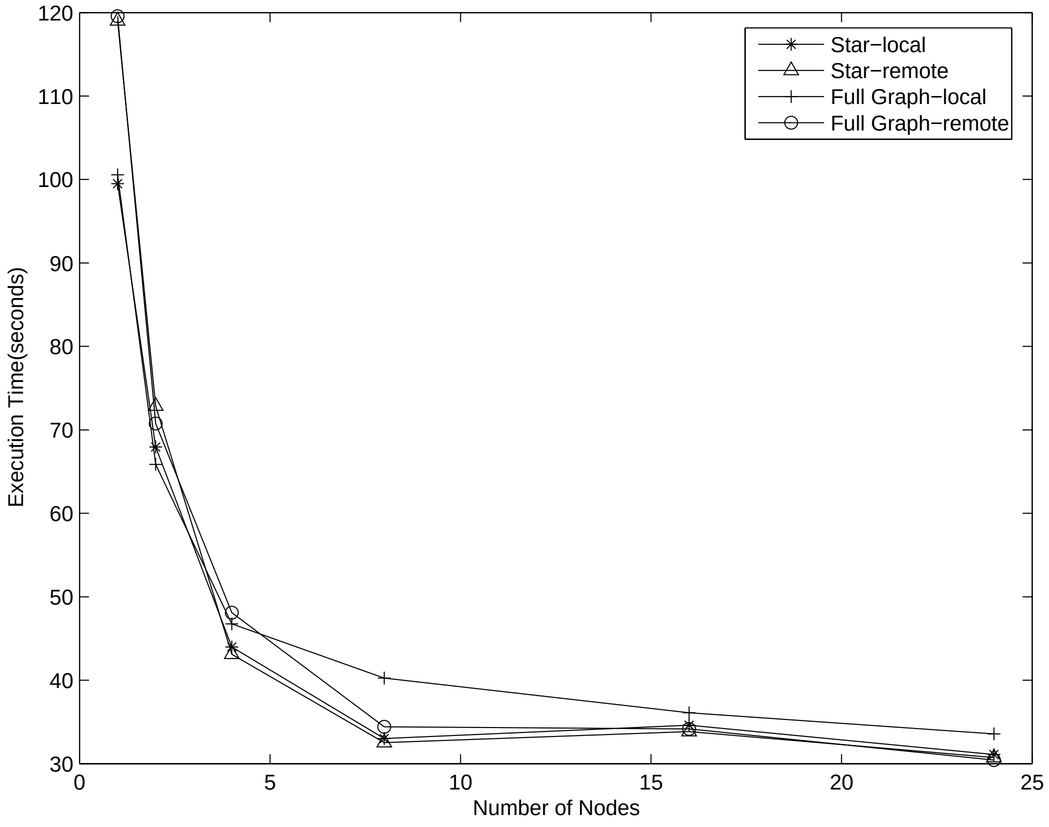


Figure 5.6: Performance for communication based tasks with two different data handling strategies. Note that having the data locally helps for smaller problem sizes since the number of remote accesses is reduced.

the original program or exposing the underlying complexities to the programmer. Similarly, one can try out other data handling strategies (like spreading the Message Boxes across multiple nodes) and adapt the program to a desired scenario.

In this Chapter we discussed the issue of programmability in large scale decentralized systems. To use the true potential these systems can possibly offer it is very important to program them efficiently. This will allow a growing range of parallel applications to be easily deployed across these systems. Our proposed architecture provides a simple interface that can be used to feed granularity information to the program at runtime. This information can then be used to determine the granularity of the application. The Message Box architecture gave us an easy way of dealing with message passing in the absence of centralized message repositories. Programs, programmers and the task scheduler are therefore freed from the message handling issues, which can be changed at anytime without affecting the rest of the system. Our real-life results, although not large scale in nature, supported our claims. We hope that this work will help in making programming for decentralized systems easier and more efficient.

Chapter 6

Related Work

In the previous chapters we studied autonomous computing in general and the issues associated with it. We suggested autonomous solutions to resource location, task scheduling and programmability. We now take a look at some of the related literature in these areas. This will help us get a better understanding of the subject and also help to compare our suggested solutions to other similar approaches.

6.1 Autonomous and Decentralized Computing

There has been a growing volume of research towards decentralizing distributed systems. Several computing systems have introduced various degrees of autonomy and decentralization in them. Even widely successful centralized systems like SETI@home [4] have introduced a hierarchical approach with BOINC [3]. In this section we take a look at some other computing systems and the kind of autonomy they have supported.

Several computing systems can be characterized by a model comprising of three layers - (i) a layer where users submit tasks, (ii) a layer where brokers/managers/schedulers schedule these tasks, and (iii) a worker layer, where the tasks actually get executed. A true ACS can be imagined to be one where every node in the system is capable of functioning as part of any of these layers. Systems such as Atlas (derived from Cilk) [8, 14] had the worker layer arranged in a tree like decentralized fashion, where workers communicated only with neighboring workers and transferred tasks through Cycle Stealing. XTremWeb, discussed in Chapter 2.4.2, supports decentralization by introducing the notion of an XTremWeb Collaborator [40], that allows grouping of a set of workers to give the system a hierarchical structure. The Javelin++ system [74] supports further autonomy by allowing the nodes in the broker layer to communicate directly between themselves, thereby further reducing the need for centralized severs. However, the system still expects broker nodes to be more reliable compared to worker nodes, therefore differentiating between them. The SuperWeb system [2] followed a similar architecture but allowed brokers and workers to be on the same node. The system also supported direct communication between the workers and the job-submitter, thereby reducing the communication bottleneck further. However, the scheduling was still done in a centralized manner. Sun Microsystem's JXTA platform [53] has a hierarchical structure where nodes are associated with groups (dispatcher group, monitor group and worker group). Nodes in the system can belong to any of these groups and can even change groups dynamically to balance the system load.

6.2 Resource Location

There has been a large volume of research addressing the issue of decentralized locating of data-centric resources in distributed systems [98, 88, 104, 54, 85, 66]. However, these efforts try to locate unique data or a unique node (address). The unique and constant value of data make it difficult to apply the same techniques for mapping computing resources. For example, with computing resources there is a limited number of resource types, and they often have a small set of values that they can assume. Using a DHT (Distributed Hash Table) based approach would save all the information in a few select nodes and make them the potential bottleneck of performance. Similarly, the value of computing resources often change rapidly, and a DHT-based approach would mean making several unnecessary updates.

However, there have been some efforts that directly address the issue of locating computing resources in distributed systems. Condor [64, 83] uses a Matchmaking algorithm where nodes periodically update their state information with server-like Matchmaker nodes that are arranged in a hierarchical fashion. While this solution is more scalable than a completely centralized approach it might not be scalable enough for global scale distributed systems. The SHARP system [36] deals with the issue of trusted resource sharing in a decentralized fashion in context of the PlanetLab [82] system. This work deals with providing resource reservation in a trusted fashion rather than the mechanism of locating these resources.

In [47] the authors address the same issue as ours and evaluated the other heuristics we discuss in this paper (random forwarding, history-based forwarding, frequency-based forwarding etc.). [97] suggested giving virtual coordinates to the resources in a multi-dimensional co-ordinate space based on their attribute values, but did not provide a way of traversing this space. In recent times, the SWORD [79, 80] system attacks a problem similar to ours and uses a Distributed Hash Table based approach for the purpose. While their system supports queries requesting a set of resources, the solution uses a hierarchical approach that is not completely decentralized. Moreover, such a solution will not work for certain scenarios (e.g. a sensor network) where it is not possible to impose a virtual topology on top of the existing network.

6.3 Task Scheduling

Task scheduling has been one of the most extensively studied research areas in distributed computing. Scheduling independent tasks across heterogeneous sets of resources is a well known problem. We differ from many of these approaches

[49, 6, 22, 61, 45] in that we are developing an autonomous scheduling strategy that does not require centralized control or knowledge for scheduling.

Several research efforts have formulated the problem of scheduling tasks across heterogeneous systems as a max-flow problem [28, 103]. However, the most popular max-flow algorithms, including Ford-Fulkerson [52] and Edmonds-Karp [31] use global information to make network-wide decisions. Golberg's algorithm [42] is closer to being autonomous but still requires a notion of *height* that depends on the total number of nodes in the network (hence it is a global information). In [93], the authors provide a parallel solution to the max-flow problem. However, their approach uses a notion of *timesteps* that are consistent across the network. This involves network-wide synchronization and is difficult to achieve in large networks. Moreover, all these techniques were designed specifically for static systems. In practice, system properties, such as node speed, bandwidth, network topology, change over time, making these techniques unsuitable.

A-FAST shares similarities with the RID (Receiver Initiated Diffusion) [69] and other similar gradient-based approaches [63, 96]. In these approaches nodes use some notion of gradient to balance their workload among their neighbors. However, they make their scheduling decisions completely based on the load at a node without taking communication abilities of the node into account. A-FAST adopts a diffusion-like approach similar to these techniques, but requests tasks based on the supply rate of a node similar to [14, 40]. This ensures that more tasks are received from nodes connected by faster link-speeds, and makes the protocol applicable to both computation and communication dominated systems. Moreover, in A-FAST all communication decisions for a pair of nodes is done independently of their remaining neighbors, reducing the synchronization requirements among nodes.

In [14, 8] variants of the Cycle Stealing technique addresses a similar problem as ours. In Cycle Stealing, a node that has exhausted all its work randomly asks its neighbors for additional work. While this approach is autonomous and works well for computation intensive applications, it requires the nodes to be arranged in a hierarchical fashion to avoid unnecessary transfer of tasks. Moreover, Cycle Stealing does not take communication time into account and does not differentiate between nodes connected by different connection speeds.

Other autonomous algorithms for independent task scheduling are presented in [11, 59, 58], which when the network is a *tree*, achieves the optimum throughput for a static network. Experiments showed that the protocol reacted quickly to changes in the network as well. However, it may not be desirable to impose a tree-structure on large networks. In [9], it is proven that the problem of finding the best tree from a given network is NP-complete, and even if one could find the best tree, there are networks for which the performance of the optimal tree is unboundedly worse than the whole networks performance. Thus, finding an autonomous optimal solution for a generic network is still open.

6.4 Programmability

There is a large body of work in programming models for parallel systems, and the idea of dynamically adapting a program with runtime information is well-known. However, there is little work specifically targeting the programming of large-scale decentralized systems. We have already mentioned that our work lies between the programming styles of MPI [70] and ATLAS [8]. ATLAS itself grew out of the Cilk System [14]. The Satin project [100] also adopted a divide and conquer approach similar to Cilk. The notion of writing programs that can adapt to resources as adaptive parallelism has been studies too in efforts such as [20, 102, 30]. Our work in Chapter 5 differs from these efforts because we provide an interface that allows the granularity of paralleism to be determined at runtime. We also provide relative priorities to the tasks. Efforts like OpenMP [78, 90] and PVM [99] are also related to our work. While the former allows programmers to include directives in the program specifying the parallelism, the latter is a software solution that treats a collection of computers as a single virtual machine. However, OpenMP is targeted at shared memory processors and is difficult to scale to a large set of resources. Even though PVM provides a unified system view, it does not provide a mechanism by which the Scheduler can feed runtime information to the program, aiding task creation. In [38] versions of a task are chosen dynamically at runtime without the programmer's interference; the technique targets a centralized pool of homogenous resources.

Recently, there have been additions to Java like RMI [86], JXTA[53] and Proactive Java [19, 46] that are intended to facilitate Parallel Computing. All these efforts allow the programmer to deal with remote resources in a local fashion. However, it is the responsibility of the programmer to deal with scheduling issues of the actual nodes. We use Java RMI as the underlying protocol in our system, but provide an architecture that hides further complexities (like scheduling and data management) from the programmer. To summarize, our work shares similarities with many of these works but provides a simple programming model that hides the complexities of the decentralized system, while allowing adaptive parallelism at runtime in such systems.

Chapter 7

Conclusions

In this dissertation we studied certain aspects of Autonomous Computing Systems, namely - resource location, task scheduling and programmability. We began with a description of Autonomous Computing Systems, their advantages and disadvantages and investigated if it was possible to design autonomous solutions to the mentioned problems that achieved performance comparable to their centralized counterparts. In this section we provide a summary of our findings and possible future work in this direction.

7.1 Summary of Findings

To address the problem of locating computing resources in an autonomous manner we proposed GUARD (Gossip Used for Autonomous Resource Detection), a protocol that uses a *gossip based distance-vector approach.* GUARD does not assume anything about the underlying system topology and is especially well suited for situations where there might be actual physical limitations on the "neighbors" of a node, e.g. sensor networks. The distance vector approach used by GUARD makes routing extremely efficient and almost optimal in terms of number of hops used (unless the table information is stale). GUARD clearly outperformed three other decentralized protocols mentioned in literature (Random, History-based and

Frequency-based) in terms of success rate (the number of queries answered) and efficiency (number of hops used). GUARD also handled multi-dimensional requests successfully, even under limitations on the routing table size. We also showed how one can put a probabilistic bound on the success rate of GUARD by controlling the gossiping frequency alone. Our experiments supported this claim and the observed results were bounded by the value predicted by our analysis. It is therefore possible to determine the required gossiping frequency based on the amount of failure that can be tolerated.

We also addressed the issue of scheduling independent homogeneous tasks in an autonomous fashion. We presented A-FAST (Autonomous Flow Approach to Scheduling Tasks), a protocol influenced by the way fluid networks function. A-FAST combined the pressure based approach of diffusion-based scheduling along with an on-demand requesting approach like Cycle Stealing. Our simulations showed that A-FAST achieved $\approx 98\%$ of the optimal steady state throughput for different topologies and systems sizes. A-FAST also adapted well to dynamically changing systems. We also suggested how A-FAST could be used to incorporate greater reliability in the system. Simulations showed A-FAST to significantly reduce task losses in faulty systems. A-FAST was also implemented and tested as part of a real system and the performance closely matched that of the estimated optimal performance for three different topologies and different system sizes. Though the test system sizes were not large, these results suggested that A-FAST is a practical protocol that can be used in a real system.

One of the biggest challenges associated with distributed computing is to provide the end user with an easy framework for programming and using it. The ease and effectiveness of programming a collection of machines is likely to play a big role in making it more popular and widespread. Decentralization makes it difficult to gauge the scale and ability of the underlying system and exposes the programmer to a much wider range of issues one has to handle. We designed and implemented a system that hides some of the underlying system complexities like scale, heterogeneity and data handling from the programmer. Our system provides a fixed and minimal set of functionalities that presents a simplified view of the system to the programmer. The programmer is expected to write adaptively parallel applications for a uniform set of homogeneous resources. Message passing was also handled in a transparent fashion where the data could be migrated across the system without requiring the programmer to be aware of it. We showed how the Scheduler can use this simple interface to deal with several system specific issues. Initial results show that the platform and interface is conducive for programming different kinds of applications and allows the programmer to be unaware of several system specific issues.

From our simulations and experimentation we could see that our proposed solutions compared with centralized solutions with respect to performance and efficiency. Though preliminary in nature these results hint at the tremendous potential Autonomous Computing Systems offer.

7.2 Future Work

Distributed Systems have been and continues to be a tremendously well studied research area. Autonomous Computing is a continuously evolving area and there is a large volume of immediate work that can be done in building better and more efficient Autonomous Systems.

One of the biggest issues we faced during our work is the absence of a test bed that is truly representative of the scale and potential of large scale systems. While efforts like Planetlab [82], FWGrid [37] etc. provide users with a collection of machines for use, the scale and ability of the nodes in these systems is very different from what one is likely to expect if one imagines a truly large scale decentralized system (e.g. Kazaa, Napster etc.). Thus, it would be really helpful to have a truly large scale decentralized computing platform for experimentation.

We have already seen large scale decentralized content-based systems like

Kazaa, Napster, Gnutella etc. However, with the exception of efforts such as SETI@home, distributed.net etc. large scale computing efforts have not yet tapped the potential of isolated machines. During our efforts to gather traces of desktop PC behavior (Appendix A), we realized that two of the biggest reasons for this is the fear of security breaches and the lack of incentives to the end user. It is therefore necessary to build systems that assure the end user of a safe execution environment in the absence of centralized control. We also need to develop applications that scale well and are able to use the large number of resources Autonomous Computing can possibly offer. This will motivate users to form computing communities where one can benefit from each other's computing power.

There are several ways in which the work presented in this dissertation can be further improved. While GUARD does an efficient job of locating individual computing resources meeting a given criteria, it does not address the issue of gathering a collection of nodes meeting a given criteria. Systems such as SWORD have addressed this issue. It would be interesting to take the completely decentralized approach of GUARD and extend it to locate a pool of nodes. GUARD currently uses a model where all nodes have a fixed gossiping frequency. One can reduce the gossiping overheads of GUARD by varying the gossiping frequency of a node adaptively.

A-FAST dealt with independent and homogeneous tasks. An immediate extension of the technique would be to support multiple task categories (especially where the tasks have different computation to communication ratios). This would allow individual users to submit different kinds of jobs and still make use of A-FAST for autonomous scheduling. Similarly, it would be nice to support task dependencies within A-FAST as that would allow a greater range of tasks to benefit from it.

Programmability of distributed systems is a vast and dormant area. Our proposed architecture was just one of the possible solutions to provide greater programming ease towards developing applications for these systems. We hope to implement a richer set of communication paradigms (like broadcast and multicast) in the future. One of the biggest challenges we faced during designing this platform was to determine a way to test the effectiveness of a programmability solution. The success of a programmability solution lies in its ease of use and the range of applications one develops on it. This can only be tested once we have large scale decentralized computing platforms and develop applications for them. We hope that with the development and deployment of these systems we would be able to find out a larger set of issues concerning them and hence build better solutions.

Work in the area of Autonomous and Decentralized Computing Systems is still in its nascent stage and far from over. We hope that the work presented in this dissertation will contribute towards making it easier to build, deploy and use large scale autonomous systems in the future.

Appendix A

Isolated Computing Resources

A.1 Introduction

In Chapter 3 we discussed detecting computing resources in an autonomous fashion so that users can benefit from these resources. The "ideal scenario" would be one where the millions of isolated computers connected by the Internet can form ad-hoc communities of their own and benefit from each other. However, there is little work characterizing these isolated machines and their behavior. A good understanding of these machines will allow us to build truly large scale autonomous computing systems that can harness the huge potential that isolated desktop machines offer. We thus conducted some profiling experiments to get an estimate of what these desktop machines are capable of. We hope that these results will help towards modeling better and more realistic large scale autonomous systems.

There have been some very useful studies of computer usage in recent times [57, 1, 15]. These studies measured the usage patterns of collections of computers in fairly homogeneous work environments. Studies like [24, 84] analyzed host-availability in enterprise and peer-to-peer systems while [60] measured the interference between communication and computation for a set of distributed machines. In [25], a detailed study of the workload and failure characteristics in the PlanetLab system is provided but these machines are part of a large academic test-bed and does not necessarily reflect the behavior of isolated desktop machines. In this chapter we try to see the extent to which these results and observations of those studies carry over to isolated personal computers. Since these isolated machines are fast emerging as a highly attractive source of computational power, it makes sense to study them separately in order to better understand how they are used. To achieve this, we monitored a set of isolated personal computers for a period of time. Particularly we tried to study the following: (i) availability of these machines (ii) their CPU usage (iii) memory usage and (iv) network usage. These parameters are often the most important ones for computing purposes and we believe a better understanding of them will lead to better techniques for utilizing these resources. We also tried to detect usage trends (i) between different machines so that we can understand if these isolated systems have common trends betwee them and (ii) the same machines at different times in order to see how volatile these systems are. Our results show that an average personal computer is quite powerful and conducive for large scale computational purposes. However, our measurements also reveal that individual users vary between themselves in their usage trends and even the same users usage patterns vary significantly with time.

A.2 Experimental Setup and Methodology

The purpose of our experiments was to monitor the way individual users use their machines. We decided to monitor the CPU utilization, main memory consumption and network utilization of these machines. We also tracked how long these machines were on and the amount of free disk space they had. For our work we restricted ourselves to machines running Windows OS. We wrote our own performance monitor that queried the WMI (Windows Management Instrumentation) database and the system registry periodically (once per second) to get the values of CPU utilization and main memory consumption. These values were then averaged for every 10 second interval. Once every minute we dumped these average values to a log file. We also recorded the amount of data transferred (both incoming and outgoing) over the network once every minute and saved this information. The whole application ran in the background as a Windows service that was automatically started every time the machine booted. Every time the program was restarted (when the system is restarted) we made a note of the free disk space in the system. The application itself used less than 1% CPU on an average and occupied 3-4 MB of the memory and did not use the network at all (these values are presented so that the effect of the performance monitor on the measurements can be better understood).

We sent out this application to over 50 different individuals (known to us and not randomly selected) for installing it on their machines. We took special care to ensure that the machines used are not part of an existing cluster and are truly isolated in nature. Of the 50 people we contacted 35 users responded and allowed us to install the application on their system. Ideally, we wanted to send out the performance monitor to a much larger group of people, but realized that a lot of people were unable (wary) to install external applications on their machines (some of these machines were part of corporate organizations and this reluctance reflects similar problems one is likely to face while dealing with these machines for actual computational purposes). Of these machines 14 were home machines, 5 machines were in corporate organizations and the remaining were located in universities across USA and India. The volunteers ran the monitoring application on their machines for a time period of 1-3 weeks, at the end of which they sent us back their log files.

A.3 Measurements

This section forms the basis of our findings where we report the measurements gathered from our experiments. Our findings are divided into three different categories - (i) basic system information, (ii) inter-user usage patterns and trends

Table A.1: Average measured value of various system parameters to give an idea of what an average isolated desktop system looks like.

Parameter	**Avg**	**25%ile**	**50%ile**	**75%ile**
on-time	61.66%	19%	65.4%	99%
disk space	37.9GB	12.5GB	22GB	69GB
RAM	483MB	480MB	512MB	512MB
free memory	208.9MB	152.2MB	202.8MB	256MB
cpu-utilization	5.49%	0.85%	4.3%	6.07%
download rate	0.329MB/min	0.060MB/min	0.134MB/min	0.276MB/min
upload rate	0.021MB/min	0.008MB/min	0.017MB/min	0.044MB/min

and (ii) intra-user patterns and trends.

A.3.1 An Average Isolated PC

Table A.1 describes the average values of some of the basic parameters we monitored. We present the on-time of these machines i.e. the percentage of time these machines were on, size of the main memory, the amount of free memory, cpu-utilization, data transfer rate (both uploading and downloading) and the amount of free hard disk space available in these systems. We have provided the mean and the 25, 50 and 75 percentile values of these parameters across all the machines we monitored.

The mean values of CPU, memory and network utilization was determined by averaging the value of all the individual readings across all the user machines. The size of the main memory and hard disk were constants that were reported once and averaged across all the machines. To determine the average on-time of these machines we calculated the difference between the last and first logged time stamp (ΔTS), the total number of time stamps recorded (N_{TS}) and the frequency between respective logs (f_{TS}). The average on-time percentage was then determined by the value $\frac{f_{TS} * N_{TS} * 100}{\Delta TS}$. For determining the average data transfer rate, we considered each interval (f_{TS}) and the data transfer that took place in it (it must be mentioned that all our measurements for data transfer rates are taken on a per-minute basis and do not reflect the bandwidth of the network since we were more interested in measuring the actual volume of data that gets transferred in these systems). The upload and download rates are provided separately.

It is interesting to note that the average computational abilities of an isolated PC are quite powerful (available $\approx 60\%$ of the times with a fairly low CPU utilization and a fair amount of free memory). This further strengthens the motivation behind commodity computing since one can easily build highly efficient and powerful systems out of these isolated machines.

A.3.2 Collective Usage Patterns

We now provide further insight into the overall view of these systems. Figures A.1-A.3 show the distribution of CPU-utilization, free-memory and data transfer rates. These values were calculated by finding the cdf (cumulative distribution of frequencies) across all the entries across all the readings we took. It is interesting to observe that the CPU is rarely introduced above 10% (94% of all our CPU utilization readings were below 10%). This hints at the tremendous amount of idle computing resource around us that we can make use of. Similarly, the data transfer rates (Figure A.3) show that seldom do these machines transfer more than a megabyte of data ($< 8\%$ of the times) and that means that the network is also highly under-utilized. The graph for fee memory is more skewed and that is largely because the available sizes of the main memory differed across these machines. Still, almost 60% of the times one can find at least 200MB of free memory in these machines.

A.3.3 Usage Intervals

We are often interested in providing dedicated service of resources to a computation, with little or no interruption (e.g. [23] only runs when the host

Table A.2: Average duration of continuous periods with given utilization constraints. Note that periods of continuous utilization above 10% are very small.

CPU Utilization	**Average Continuous Duration**
$< 1\%$	15.20 mins
$< 5\%$	29.44 mins
$< 10\%$	50.46 mins
$> 1\%$	5.30 mins
$> 5\%$	3.60 mins
$> 10\%$	3.20 mins

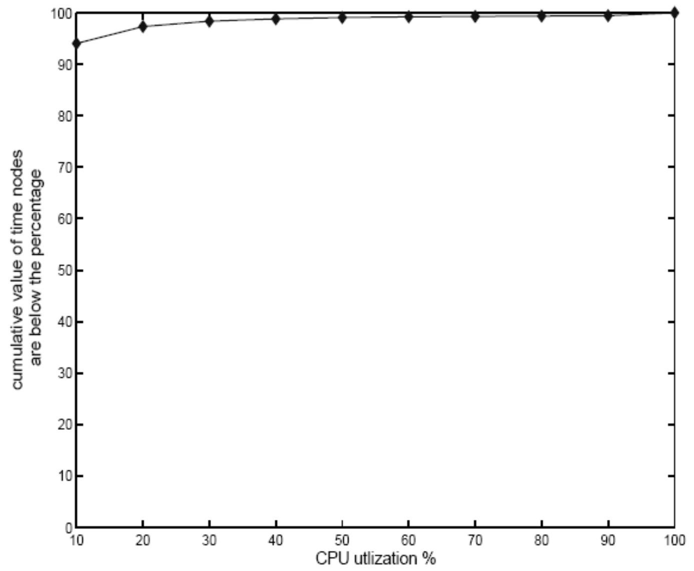


Figure A.1: Cumulative distribution of time node utilization is below a certain percentage.

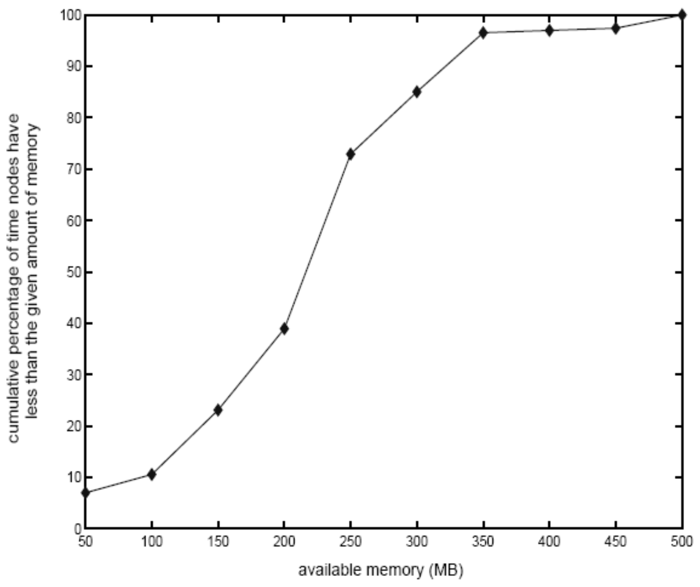


Figure A.2: Cumulative distribution of time nodes have below a given amount of free memory.

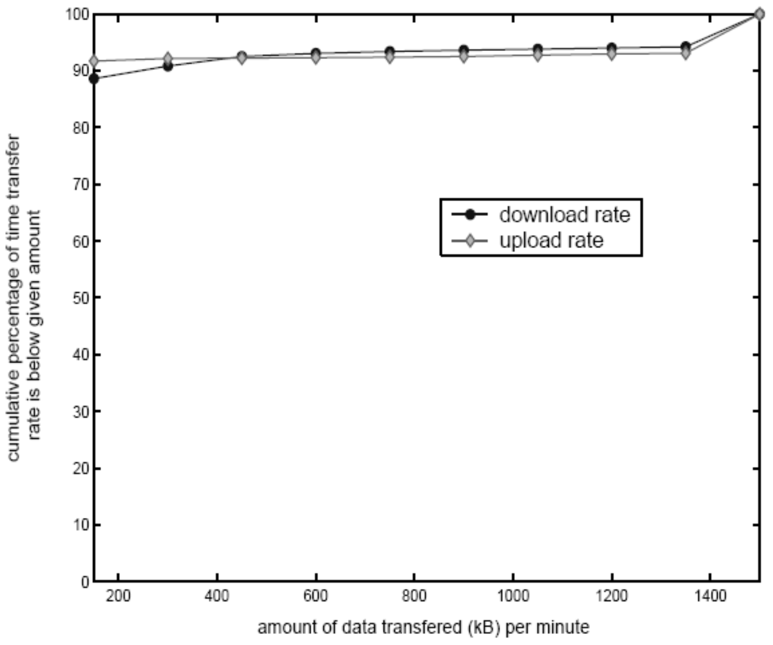


Figure A.3: Cumulative distribution of time nodes transfer below a given amount of data per minute. Both the download and upload rates (in kB/min) are given.

Table A.3: The mean deviation in user behavior for a given interval. This shows how much the values vary for an individual user with time.

Duration	CPU Use	Free Memory	Download/min	Upload/min
10 mins	3.47%	23.78MB	338.15kB	877.40kB
1 hour	4.53%	21.40B	351.60kB	891.70kB
10 hours	4.93%	20.46MB	161kB	356.32kB

machine is completely idle). We thus tried to determine the average duration a node remained below a particular utilization continuously, without any interruption. Table A.2 show the value of intervals for which a node was completely below a particular utilization. For each utilization constraint, we observed the log file to see how many continuous entries adhered to the desired constraint. The values of these durations were then averaged across all the entries of all the users. One can see that on an average a user can get almost 50 minutes of dedicated time if he wants the machine to have less than 10% CPU utilization. This is very important for jobs with stringent QoS requirements. Even for requirements of less than 1% utilizations the average uninterrupted interval length is more than 15 mins. We have also shown that, when used, a machine doesn't get used for too long at a stretch. Even when a machine has a CPU-utilization $> 10\%$, the period lasts less than 3 and a half minutes on an average. Therefore a task has to wait for only small periods before the machine is under-utilized again.

A.3.4 Inter-user Usage Patterns

In the previous section we presented an overall view of what an isolated system is capable of and how it gets used. However, it is likely that individual users use their systems differently and it is important to investigate if we can generalize these users as a single uniform group. That is what we try to do in this section.

Figures A.4-A.7 shows how the values of the various usage parameters, presented in the previous section, differed across the individual users. We calcu-

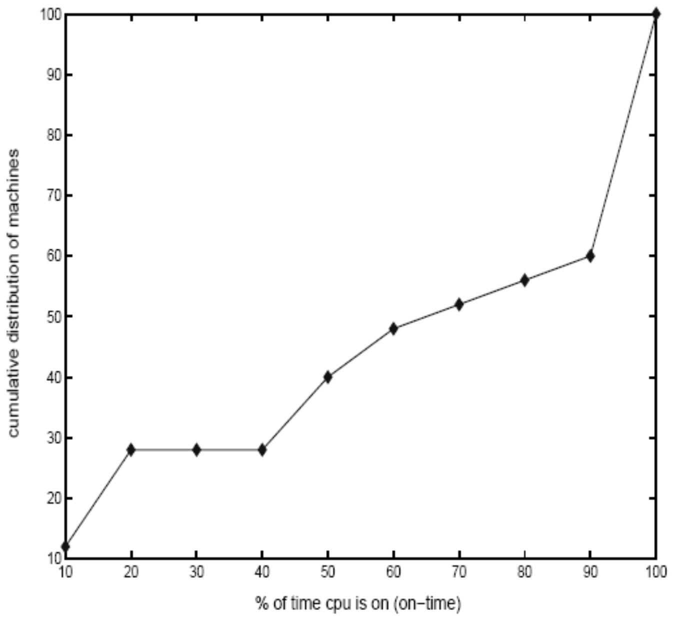


Figure A.4: Distribution showing trend of *on-time* (the period for which a node is continuously on) across the various users.

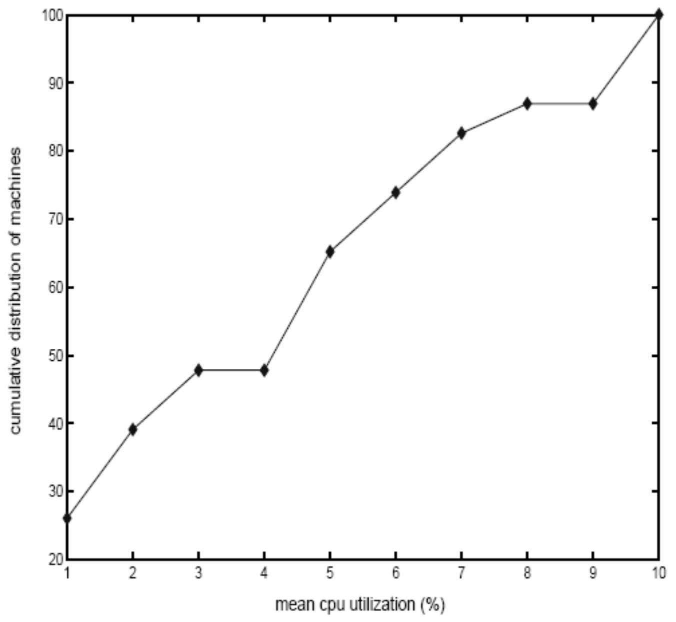


Figure A.5: Distribution showing trend of *CPU utilization* across the various users.

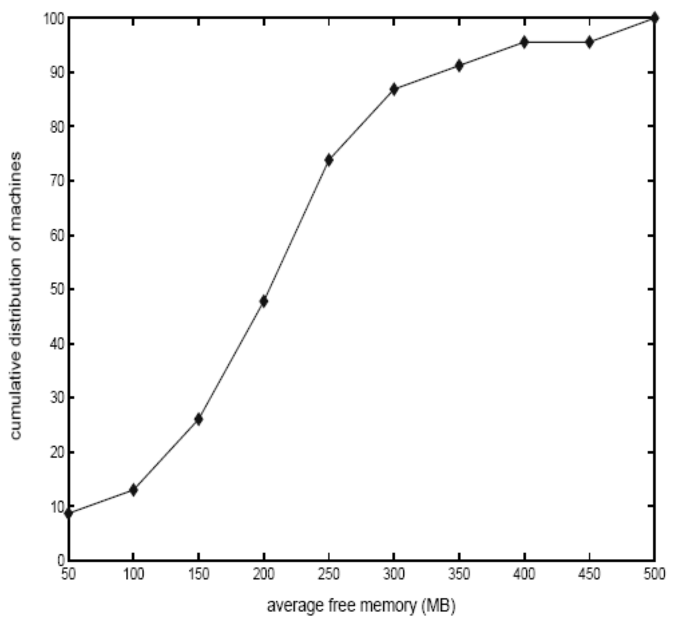


Figure A.6: Distribution showing trend of *Available free memory* across the various users.

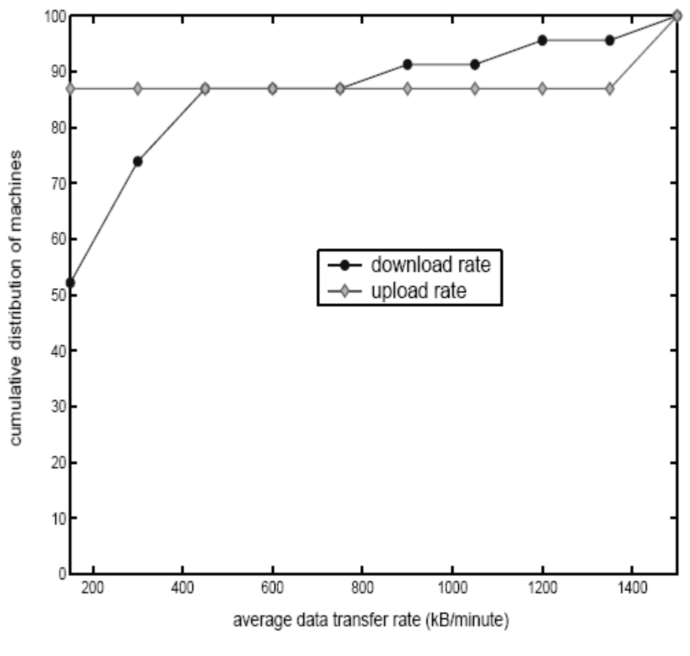


Figure A.7: Distribution showing trend of *Data transfer rates* (both upload and download) across the various users.

lated the average value of these parameters across all the readings for each user separately and plotted how the mean values differed across users. This gives us an idea of how different individual users are in terms of their system usage.

One can observe that while data transfer patterns remain quite constant across most users, other parameters (on-time, cpu-utilization and free-memory) vary widely across users. The graph of on-times show that almost 40% of the users leave their machines on all the time (100% on-time). That coupled with the fact that more than 85% users have a mean CPU-utilization of less than 10% (Figure A.5)) shows the tremendous potential of these isolated machines.

A.3.5 Intra-user Usage Patterns

In the previous two sections we showed how an average isolated personal computer functions and the differences between the average performance of these machines. We now show how the performance of these same machines change with time. Users are likely to use their machines differently at different times and this section is intended to give an idea of how different user behavior is over time.

To determine the variation in user behavior we divided each users behavior over a given amount of time and found the standard deviation of the performance across the different intervals. This would reflect how much the users behavior differs across different intervals of the given duration. We then averaged these values across all the users. The values are given in Table A.3 and reflect the mean deviation in an users behavior for the given interval. We calculated these values for intervals of length 10 minutes, 1 hour and 10 hours.

It is interesting to observe that there exists a fairly strong variation in user behavior in terms of CPU-utilization (considering the mean CPU utilization in our findings to be around 6.5%) and data transfer rates over time, even for individual users. The usage of memory however remains more constant. This shows that a machine is unlikely to perform in a fixed way and algorithms and applications built for commodity computing should take this volatility into account.

In this chapter we tried to study the behavior of isolated desktop computers in order to better understand their abilities and to use them more efficiently for the purpose of commodity computing. We found that an average isolated PC is quite powerful (Section A.3.1) and conducive for computing purposes. We also saw that individual users show different usage trends between themselves (Section A.3.4) and at different times (Section A.3.5). These variations, if taken into account, will help in designing better scheduling techniques. This hints that one should definitely try to tap the potential of the millions of isolated PCs across the world but at the same time design algorithms and application that take into account their variability and volatility.

Appendix B

Ongoing Work - Value-centric Scheduling

In Section 5.2.2 we mentioned that it is possible for users to prioritize individual tasks. The scheduler can then try to schedule tasks based on the user assigned priority. This allows users to set the relative importance of their jobs by assigning a certain degree of importance to them. Recent efforts have modeled computational grids as a marketplace [18, 92, 26, 94] where users "pay" for the computational resources that they use. As part of our ongoing research we are considering various strategies for scheduling in such a scenario where users assign costs to their jobs. This chapter mentions some of our suggested approaches.

B.1 Problem Definition

In this section we formally define the problem we are dealing with and the various assumptions that we make about the underlying system.

Our system comprises of a single scheduler (S). A job, j, in the system is defined as a 2-tuple $< f(t), l >$ where $f(t)$ is a function that denotes the price of the job (the value the system gets on completing the job) and is a function of the turnaround time (t) of the job; l is the length of the job in terms of number of instructions (unlike previous works [51, 27] which assumed a single resource or homogenous resources we cannot represent the length of the job in terms of time). The scheduler has a pool of N processing units under its control where $p_1, p_2, ..., p_N$ denotes the speed of these processors in terms of the number of instructions that they can execute in a unit of time. Without any loss of generality we assume that $p_1 \geq p_2 \geq ... \geq p_N$. A total of K jobs $(j_1, j_2, ..., j_K)$ arrive in an online fashion (i.e. S has no idea about the arrival time of the jobs) and get submitted to the scheduler. Once a job arrives S can either assign it to one of the idle processing units p_i, or it can choose to put it in a wait queue. If assigned to a processor the job begins execution immediately. In our model jobs are non-preemptive i.e. once a job gets assigned to a processor, the job runs till completion on that processor. We do not consider failures in our system. Each processor can only run one job at a time and each job can run on only one processor at a time (i.e. jobs are not divisible or parallelizable). All jobs, once submitted to the system, have to be completed and cannot be discarded before completion. The objective of the scheduler is to maximize the value/gain of the system (i.e. *maximize* $\sum_{i=1}^{K} f(t_i)$, where t_i is the total time job i spends in the system.

Ideally speaking users should be able to set any function $f_i(t)$ of their choice for each job j_i. However, traditionally people have used functions that penalize the systems (for delay) linearly with time [51, 27]. While some work has tried to deal with step functions that changes the yield of a job discretely, a growing volume of research has accepted linearly decaying yield as a standard pricing strategy for such market based systems even though such a pricing strategy doesn't address several issues (e.g. partial payment for partial completion or providing additional incentives for completing a job quickly).

For the rest of this chapter we assume that every job j_i's associated value-function $f_i(t)$ is given by $f_i(t) = a_i$ - $b_i.t$. Thus every job j_i has a basic value a_i that the system get's on completing the job and has an associated penalty given by $b_i.t$ that the systems pays to the user[1]. The constant b_i is called the *penalty coefficient* of the job and is a reflection of how critical a job is to delay. Since the system has to complete any job that it accepts, the order and time in which the jobs get completed doesn't affect the earning from the a values of the jobs (since they are time invariant). Thus, in order to maximize the gain, one has to minimize the overall penalty paid by the system (i.e. *maximize* $\sum_{i=1}^{K} f(t_i) \equiv$ *minimize* $\sum_{i=1}^{K} b_i.t_i$). It is therefore possible to think of each job, j_i, as a 2-tuple given by $< b_i, l_i >$. This is the formulation we will use for the rest of this chapter.

B.2 Value-centric Scheduling

The problem described in the previous section is very similar to the total weighted completion time (TWCT) problem [7]. In TWCT, every job is characterized by a 2-tuple $< b, t >$ where b is a penalty coefficient similar to ours and t is the duration of the job. TWCT deals with only one processing unit and the objective of the system is to minimize $\sum b_i.C_i$ where C_i is the total completion time of the job. The off-line instance of TWCT itself is NP-hard and it is easy to show that every instance of a TWCT problem can be reduced to our scenario (by setting $N = 1$ and setting the l_i of each job to t_i). It is therefore unlikely to come up with optimal scheduling strategies. In this section we suggest 3 different prioritizing heuristics and 3 different buffering techniques and discuss their individual properties.

B.2.1 Prioritizing Techniques

We now discuss different prioritizing techniques and our motivation for trying each one out. It must be mentioned that it is possible to come up with situations where each technique will outdo the others and therefore there is no clear winner.

[1]It is therefore possible for the system to actually pay for a job if $b_i.t > a_i$.

Technique T1

In this strategy we prioritize all the jobs in the system based on their b/l value i.e. the job in the task queue with the highest value of b/l gets the fastest available processor. A higher value of l implies that the job will be running for a longer period of time and it is therefore possible that a task of greater penalty would enter the system and suffer due to the unavailability of the resource(s). The objective of this approach is thus to give jobs with higher penalties greater precedence while discounting them by their expected time of completion (proportional to l). This is similar to the approach used in [51].

Technique T2

In this technique we prioritize the jobs based on their expected expected net penalties i.e. based on their values of $b.l$. It is easy to see that this is quite the opposite of T1 and tries to assign the fastest processor to the job which is likely to cost *us* the most. While such a technique would do well in systems where most jobs are long running with infrequent arrivals, it would be a bad strategy to use in systems where lots of high priority short jobs are present.

Technique T3

None of the strategies described till now consider the heterogeneity of the processing units explicitly. The amount of disparity between the various processing units can play a significant role on the scheduling decision and this strategy tries to address that issue.

This technique begins by assigning random unique processors to the jobs (if there are more jobs than processors, one can imagine an infinite number of additional processors with $p = 0$). The jobs are then sorted based on the condition: $C(j_m, p_x) + C(j_n, p_y) > C(j_n, p_x) + C(j_m, p_y)$, where $C(j_m, p_x)$ is the cost we pay for running job j_m on processor p_x (for our system $C(j_m, p_x) = b_m.l_m/p_x$. Thus in this technique, we test if swapping two jobs is more profitable than their present assignment and if so we do it. It is interesting to see that unlike the previous three techniques *this technique is not tied to the pricing policy of linear penalty that we have adopted and can be used in any generalized pricing scenario.*

B.2.2 Buffering Techniques

In most existing research in scenarios similar to ours, if there is a job in the wait queue and a processing unit that is available, then then scheduler always assigns the job to the processing unit that is best suited for it. While this seems like an obvious thing to do, it might not be the best strategy. In an online scenario where jobs are continuously entering the system, it might make sense to wait and buffer up tasks even though there is an available processor. This might help make better decisions if more valuable future jobs come and since they can then get an available resource. Such a buffering strategy would mean starving existing jobs in expectation of more lucrative deals for the available processing units. It must be mentioned though, that the buffering techniques only decide whether a task should wait or not and does not affect the prioritization of a task. Thus they have to be used along with some scheduling technique (like the ones described above). In this section we discuss three different buffering strategies that we adopted in our work.

Technique B1

Our first buffering technique is the simplest one to implement. We define a parameter B and postpone all scheduling decisions until we have at least B jobs in the job queue. While this is easy to implement and gives the scheduler a broader window to make its decisions, it is easy to see that in a system where jobs arrive infrequently, such a buffering technique would lead to large volumes of unnecessary starvation.

Technique B2

In this technique, once a job is assigned to a processor, we check to see if any of the already occupied faster processors are better suited for the job and if so we make free the processor the job has currently been assigned to and make it wait for the higher priority processor to free up. Each processor p_i also has a flag f_i that denotes if the processor has already been booked for a future job using this buffering technique (this is done to prevent multiple jobs from waiting for the same processor). Formally speaking, if a job j_i is assigned to processor p_x we search for the first available p_y $(y < x)$ such that ($(f_y = \mathit{false})$ && ($b_i.(l_i + l')/p_y < b_i.l_i/p_x)$), where l' is the number of instructions left for the job currently running on p_y to finish. If we find such a y we set the value of $f_y = true$.

Since this strategy only makes a task wait if it can do better than currently running it - it should always do well. However, it is possible that while a task waits for a processor to free up another higher priority task enters the system and occupies the processor, thereby making the wait futile (a new task clears all the values of f_is and the assignments are done afresh). Thus it is possible for this strategy too to have negative effects as well.

Technique B3

Our third buffering technique tries to achieve the benefits of B1 while preventing the possibility of unnecessary starvation due to infrequent job arrivals. We define a parameter F that denotes the acceptable fraction of loss for a job due to waiting. When a job j_i gets assigned to a processor p_x, it is made to wait an additional amount of time determined by $F.l_i/p_x$. The job thus waits for an additional amount of time that is determined by the fraction of loss the system is willing to suffer. Once a job's wait is over it is considered for scheduling along with all other jobs with higher priority that are waiting in the system. It must be noted that a job might get assigned to a processor, even if it's wait time is not over, provided the wait of a job with lower priority is over. This is done to ensure that at no point does a lower priority job ever get assigned to a processor while there is a more critical job waiting.

As mentioned earlier, this technique tries to give the scheduler a broader window of jobs to make it's decision while putting a bound on the amount of loss the system suffers when waiting for this window to build up.

In this chapter we took a look at some of our ongoing work in scheduling tasks based on user given priorities. We dealt with a specific group of non-preemptive tasks where buffering techniques might be very useful. However, it is too early to comment on the usefulness of these techniques and as value functions evolve and the nature of market-based computational grids are better understood, we look forward to verifying the usefulness of these techniques further.

Bibliography

[1] Anurag Acharya, Guy Edjlali, and Joel Saltz. The Utility of Exploiting Idle Workstations for Parallel Computation. In *SIGMETRICS '97: Proceedings of the 1997 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 225–234, New York, NY, USA, 1997. ACM Press.

[2] Albert D. Alexandrov, Maximilian Ibel, Klaus E. Schauser, and Chris J. Scheiman. SuperWeb: Research Issues in Java-based Global Computing. *Concurrency: Practice and Experience*, 9(6):535–553, 1997.

[3] David P. Anderson. BOINC: A System for Public-Resource Computing and Storage. In *GRID*, pages 4–10, 2004.

[4] David P. Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, and Dan Werthimer. SETI@home: An Experiment in Public-resource Computing. *Commun. ACM*, 45(11):56–61, 2002.

[5] D. Andresen and T. McCune. Towards a Hierarchical Scheduling System for Distributed WWW Server Clusters. In *Proceedings of the Seventh International Symposium on High Performance Distributed Computing (HPDC-7)*, July 1998.

[6] D. Andresen and T. McCune. Towards a Hierarchical Scheduling System for Distributed WWW Server Clusters. In *HPDC '98: Proceedings of the The Seventh IEEE International Symposium on High Performance Distributed Computing*, page 301, Washington, DC, USA, 1998. IEEE Computer Society.

[7] Ivan D. Baev, Waleed Meleis, and Alexandre E. Eichenberger. An Experimental Study of Algorithms for Weighted Completion Time Scheduling. *Algorithmica*, 33(1):34–51, 2002.

[8] J. Baldeschwieler, R. Blumofe, and E. Brewer. ATLAS: An Infrastructure for Global Computing. *In Proceedings of the Seventh ACM SIGOPS European Workshop on System Support for Worldwide Applications*, 1996.

[9] C. Banino, O. Beaumont, A. Legrand, and Y. Robert. Scheduling Strategies for Master-Slave Tasking on Heterogeneous Processor Grids. Technical Report RR2002-12, ENS-Lyon, LIP, 2002.

[10] A. Baratloo, M. Karaul, Z. M. Kedem, and P. Wijckoff. Charlotte: Metacomputing on the Web. *Future Gener. Comput. Syst.*, 15(5-6):559–570, 1999.

[11] O. Beaumont, L. Carter, J. Ferrante, A. Legrand, and Y. Robert. Bandwidth-centric Allocation of Independent Task on Heterogeneous Platforms. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'02), Fort Lauderdale, Florida*, April 2002.

[12] BitTorrent. http://www.bittorrent.com/.

[13] R. Blumofe, C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall, and Y. Zhou. Cilk: An Efficient Multithreaded Runtime System. In *Proceedings of the 5th Symposium on Principles and Practice of Parallel Programming*, 1995.

[14] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An Efficient Multithreaded Runtime System. In *PPOPP '95: Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 207–216, New York, NY, USA, 1995. ACM Press.

[15] William J. Bolosky, John R. Douceur, David Ely, and Marvin Theimer. Feasibility of a Serverless Distributed File System Deployed on an Existing set of Desktop PCs. *SIGMETRICS Perform. Eval. Rev.*, 28(1):34–43, 2000.

[16] George Bosilca, Aurelien Bouteiller, Franck Cappello, Samir Djilali, Gilles Fedak, Cecile Germain, Thomas Herault, Pierre Lemarinier, Oleg Lodygensky, Frederic Magniette, Vincent Neri, and Anton Selikhov. MPICH-V: Toward a Scalable Fault Tolerant MPI for Volatile Nodes. In *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–18, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[17] BSP Worldwide. http://www.bsp-worldwide.org/.

[18] Rajkumar Buyya, David Abramson, and Jonathan Giddy. A Case for Economy Grid Architecture for Service Oriented Grid Computing. In *IPDPS '01: Proceedings of the 10th Heterogeneous Computing Workshop, HCW (Workshop 1)*, page 20083.1, Washington, DC, USA, 2001. IEEE Computer Society.

[19] Denis Caromel, Wilfried Klauser, and Julien Vayssière. Towards Seamless Computing and Metacomputing in Java. *Concurrency: Practice and Experience*, 10(11–13):1043–1061, 1998.

[20] Nicholas Carriero, Eric Freeman, David Gelernter, and David Kaminsky. Adaptive Parallelism and Piranha. *Computer*, 28(1):40–49, 1995.

[21] H. Casanova, A. Legrand, D. Zagorodnov, and F. Berman. Heuristics for Scheduling Parameter Sweep Applications in Grid Environments. In *Proceedings of the 9th Heterogeneous Computing Workshop (HCW'00)*, May 2000.

[22] Henri Casanova, Dmitrii Zagorodnov, Francine Berman, and Arnaud Legrand. Heuristics for Scheduling Parameter Sweep Applications in Grid Environments. In *HCW '00: Proceedings of the 9th Heterogeneous Computing Workshop*, page 349, Washington, DC, USA, 2000. IEEE Computer Society.

[23] Andrew Chien, Brad Calder, Stephen Elbert, and Karan Bhatia. Entropia: Architecture and Performance of an Enterprise Desktop Grid System. *J. Parallel Distrib. Comput.*, 63(5):597–610, 2003.

[24] J. Chu, K. Labonte, and B. Levine. Availability and Locality Measurements of Peer-to-peer File Systems. *Proceedings of ITCom: Scalability and Traffic Control in IP Networks*, july 2003.

[25] B. Chun and A. Vahdat. Workload and Failure Characterization on a Large-scale Federated Testbed. Technical Report IRB-TR-03-040, Intel Research Berkeley, November 2003.

[26] Brent N. Chun, Philip Buonadonna, Alvin AuYoung, Chaki Ng, David C. Parkes, Jeffrey Shneidman, Alex C. Snoeren, and Amin Vahdat. Mirage: A Microeconomic Resource Allocation System for Sensornet Testbeds. In *2nd IEEE Workshop on Embedded Networked Sensors (EmNetS-II)*, May 2005.

[27] Brent N. Chun and David E. Culler. User-Centric Performance Analysis of Market-Based Cluster Batch Schedulers. In *CCGRID '02: Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*, page 30, Washington, DC, USA, 2002. IEEE Computer Society.

[28] T. Cormen, C. Leiserson, and R. Rivest. Introduction to Algorithms, MIT Press. 1990.

[29] Distributed.net. http://www.distributed.net.

[30] Guy Edjlali, Gagan Agrawal, Alan Sussman, and Joel H. Saltz. Data Parallel Programming in an Adaptive Environment. In *IPPS '95: Proceedings of the 9th International Symposium on Parallel Processing*, pages 827–832, Washington, DC, USA, 1995. IEEE Computer Society.

[31] Jack Edmonds and Richard M. Karp. Theoretical Improvements in the Algorithmic Efficiency for Network Flow problems. *Journal of the ACM (JACM)*, 19, 1972.

[32] S. Flynn Hummel, J. Schmidt, R. Uma, and J. Wein. Load-Sharing in Heterogeneous Systems via Weighted Factoring. In *Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'96)*, Jun 1996.

[33] I. Foster and C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, Summer 1997.

[34] Ian Foster and Carl Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2), Summer 1997.

[35] FreeNet. http://freenet.sourceforge.net/.

[36] Yun Fu, Jeffrey Chase, Brent Chun, Stephen Schwab, and Amin Vahdat. SHARP: An Architecture for Secure Resource Peering. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 133–148, New York, NY, USA, 2003. ACM Press.

[37] FWGrid. http://fwgrid.ucsd.edu/.

[38] Kang Su Gatlin and Larry Carter. Architecture-cognizant Divide and Conquer Algorithms. In *Supercomputing '99: Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)*, page 25, New York, NY, USA, 1999. ACM Press.

[39] C. Germain, G. Fedak, V. Néri, and F. Cappello. Global Computing Systems. *Lecture Notes in Computer Science*, 2179, 2001.

[40] Cile Germain, Vincent, Gilles Fedak, and Franck Cappello. XtremWeb: Building an Experimental Platform for Global Computing. In *GRID '00: Proceedings of the First IEEE/ACM International Workshop on Grid Computing*, pages 91–101, London, UK, 2000. Springer-Verlag.

[41] Gnutella. http://www.gnutella.com.

[42] Andrew V. Goldberg. *Efficient Graph Algorithms for Sequential and Parallel Computers.* PhD thesis, Department of Electrical Engineering and Computer Science, MIT, 1987.

[43] James N. Gray. An Approach to Decentralized Computer Systems. *IEEE Trans. Softw. Eng.*, 12(6):684–692, 1986.

[44] T. Hagerup. Allocating Independent Tasks to Parallel Processors: An Experimental Study. *Journal of Parallel and Distributed Computing*, 47, 1997.

[45] Torben Hagerup. Allocating Independent Tasks to Parallel Processors: An Experimental Study. *J. Parallel Distrib. Comput.*, 47(2):185–197, 1997.

[46] Fabrice Huet, Denis Caromel, and Henri E. Bal. A High Performance Java Middleware with a Real Application. In *Proceedings of the Supercomputing conference*, Pittsburgh, Pensylvania, USA, November 2004.

[47] Adriana Iamnitchi and Ian T. Foster. On Fully Decentralized Resource Discovery in Grid Environments. In *GRID '01: Proceedings of the Second International Workshop on Grid Computing*, pages 51–62, London, UK, 2001. Springer-Verlag.

[48] O. H. Ibarra and C. E. Kim. Heuristic Algorithms for Scheduling Independent Tasks on non-identical processors. *Journal of the ACM (JACM)*, 24(2), 1997.

[49] Oscar H. Ibarra and Chul E. Kim. Heuristic Algorithms for Scheduling Independent Tasks on Nonidentical Processors. *J. ACM*, 24(2):280–289, 1977.

[50] Chalermek Intanagonwiwat, Ramesh Govindan, and Deborah Estrin. Directed Diffusion: A Scalable and Robust Communication Paradigm for Sensor Networks. In *Mobile Computing and Networking*, pages 56–67, 2000.

[51] David E. Irwin, Laura E. Grit, and Jeffrey S. Chase. Balancing Risk and Reward in a Market-Based Task Service. In *HPDC '04: Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing (HPDC'04)*, pages 160–169, Washington, DC, USA, 2004. IEEE Computer Society.

[52] L. R. Ford Jr. and D. R. Fulkerson. Flow in Networks, Princeton University Press. 1962.

[53] Project JXTA. http://www.jxta.org.

[54] M. Frans Kaashoek and David R. Karger. Koorde: A Simple Degree-optimal Distributed Hash Table. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, 2003.

[55] Kazaa Networks. http://www.kazaa.com.

[56] Derrick Kondo, Henri Casanova, Eric Wing, and Francine Berman. Models and Scheduling Mechanisms for Global Computing Applications. In *IPDPS '02: Proceedings of the 16th International Parallel and Distributed Processing Symposium*, page 216, Washington, DC, USA, 2002. IEEE Computer Society.

[57] Derrick Kondo, Michela Taufer, Charles L. Brooks III, Henri Casanova, and Andrew A. Chien. Characterizing and Evaluating Desktop Grids: An Empirical Study. In *IPDPS*, 2004.

[58] B. Kreaseck, L. Carter, H. Casanova, J. Ferrante, and S. Nandy. Interference-Aware Scheduling. In *International Journal of High Performance Computing Applications (IJHPCA)*, 2006.

[59] B. Kreaseck, L. Carter, H. Casanova, and J.Ferrante. Autonomous Protocols for Bandwidth-Centric Scheduling of Independent-task Applications. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'03), Nice, France*, April 2003.

[60] Barbara Kreaseck, Larry Carter, Henri Casanova, and Jeanne Ferrante. On the Interference of Communication on Computation in Java. In *Proceedings of the 3rd International Workshop on Performance Modeling, Evaluation and Optimization on Parallel and Distributed Systems (PMEO-PDS'04), Santa Fe, New Mexico*, April 2004.

[61] Clyde P. Kruskal and Alan Weiss. Allocating Independent Subtasks on Parallel Processors. *IEEE Trans. Softw. Eng.*, 11(10):1001–1016, 1985.

[62] C.P. Kruskal and A. Weiss. Allocating Independent Subtasks on Parallel Processors. *IEEE Transactions on Software Engineering*, 11, 1984.

[63] Frank C. H. Lin and Robert M. Keller. The Gradient Model Load Balancing Method. *IEEE Trans. Softw. Eng.*, 13(1):32–38, 1987.

[64] Michael Litzkow, Miron Livny, and Matthew Mutka. Condor - A Hunter of Idle Workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, June 1988.

[65] M. Maheswaran, S. Ali, H. J. Siegel, D. Hensgen, and R. Freund. Dynamic Matching and Scheduling of a Class of Independent Tasks onto Heterogeneous Computing Systems. In *8th Heterogeneous Computing Workshop (HCW'99)*, pages 30–44, Apr. 1999.

[66] Dahlia Malkhi, Moni Naor, and David Ratajczak. Viceroy: A Scalable and Dynamic Emulation of the Butterfly. In *PODC '02: Proceedings of the twenty-first annual symposium on Principles of distributed computing*, pages 183–192, New York, NY, USA, 2002. ACM Press.

[67] Max-Flow. http://elib.zib.de/pub/Packages/mathprog/maxflow/index.html.

[68] Great Internet Mersenne Prime Search (GIMPS). http://www.mercenne.com.

[69] Willebeek-LeMair M.H. and A.P Reeves. Strategies for Dynamic Load Balancing on Highly Parallel Computers. In *Parallel and Distributed Systems, IEEE Transactions*, 1993.

[70] The MPI-Forum. http://www.mpi-forum.org/docs/docs.html.

[71] Sagnik Nandy, Larry Carter, and Jeanne Ferrante. A-FAST: Autonomous Flow Approach to Scheduling Tasks. In *HiPC*, pages 363–374, 2004.

[72] Sagnik Nandy, Larry Carter, and Jeanne Ferrante. GUARD: Gossip Used for Autonomous Resource Detection. In *IPDPS*, 2005.

[73] Napster. http://www.napster.com.

[74] Michael O. Neary, Sean P. Brydon, Paul Kmiec, Sami Rollins, and Peter Cappello. Javelin++: Scalability Issues in Global Computing. *Concurrency: Practice and Experience*, 12(8):727–753, 2000.

[75] Michael O. Neary, Bernd O. Christiansen, Peter Cappello, and Klaus E. Schauser. Javelin: Parallel Computing on the Internet. *Future Gener. Comput. Syst.*, 15(5-6):659–674, 1999.

[76] Network Emulator. http://clarinet.u-strasbg.fr/nem/.

[77] Licnio Oliveira, Lus Lopes, and Fernando Silva. P^3: Parallel Peer to Peer - An Internet Parallel Programming Environment. In *International Workshop on Peer-to-peer Computing*, 2002.

[78] OpenMP. http://www.openmp.org/.

[79] D. Oppenheimer, J. Albrecht, D. Patterson, and A. Vahdat. Distributed Resource Discovery on PlanetLab with SWORD. In *First Workshop on Real, Large Distributed Systems (WORLDS04)*, December 2004.

[80] D. Oppenheimer, J. Albrecht, D. Patterson, and A. Vahdat. Design and Implementation Tradeoffs for Wide-Area Resource Discovery. In *HPDC '05: Proceedings of the 14th IEEE International Symposium on High Performance Distributed Computing (HPDC'05)*, July 2005.

[81] Charles Perkins and Pravin Bhagwat. Highly Dynamic Destination-Sequenced Distance-Vector Routing (DSDV) for Mobile Computers. In *ACM SIGCOMM'94 Conference on Communications Architectures, Protocols and Applications*, pages 234–244, 1994.

[82] PlanetLab. http://www.planet-lab.org/.

[83] Rajesh Raman, Miron Livny, and Marv Solomon. Matchmaking: An Extensible Framework for Distributed Resource Management. *Cluster Computing*, 2(2):129–138, 1999.

[84] Geoffrey Voelker Ranjita Bhagwan, Stefan Savage. Understanding Availability. In *The 2nd International Workshop on Peer-to-Peer Systems (IPTPS)*, February 2003.

[85] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard M. Karp, and Scott Shenker. A Scalable Content-addressable Network. In *SIGCOMM*, pages 161–172, 2001.

[86] RMI. http://java.sun.com/products/jdk/rmi/.

[87] A. Rosenberg. Sharing Partitionable Workloads in Heterogeneous NOWs: Greedier Is Not Better. In *Proceedings of the IEEE International Conference on Cluster Computing (Cluster'01), Newport Beach, California*, October 2001.

[88] Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. In *Middleware 2001: Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, pages 329–350, London, UK, 2001. Springer-Verlag.

[89] Luis F. G. Sarmenta and Satoshi Hirano. Bayanihan: Building and Studying Web-based Volunteer Computing Systems using Java. *Future Gener. Comput. Syst.*, 15(5-6):675–686, 1999.

[90] Alex Scherer, Honghui Lu, Thomas Gross, and Willy Zwaenepoel. Transparent adaptive parallelism on NOWs using OpenMP. In *PPoPP '99: Proceedings of the seventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 96–106, New York, NY, USA, 1999. ACM Press.

[91] SETI@home. http://setiathome.ssl.berkeley.edu, 2001.

[92] Jahanzeb Sherwani, Nosheen Ali, Nausheen Lotia, Zahra Hayat, and Rajkumar Buyya. Libra: A Computational Economy-based Job Scheduling System for Clusters. *Softw. Pract. Exper.*, 34(6):573–590, 2004.

[93] Y. Shiloach and U. Vishkin. An $O(n^2 log\, n)$ Parallel MAX-FLOW Algorithm. *Journal of Algorithms*, 1982.

[94] Jeffrey Shneidman, Chaki Ng, David Parkes, Alvin AuYoung, Alex C. Snoeren, Amin Vahdat, and Brent N. Chun. Why Markets Could (But Don't Currently) Solve Resource Allocation Problems in Systems. In *10th USENIX Workshop on Hot Topics in Operating Systems (HotOS-X)*, June 2005.

[95] John F. Shoch and Jon A. Hupp. The Worm Programs: Early Experience with a Distributed Computation. *Commun. ACM*, 25(3):172–180, 1982.

[96] W. Shu and L. V. Kale. A Dynamic Scheduling Strategy for the Chare-Kernel System. In *Supercomputing '89: Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, pages 389–398, New York, NY, USA, 1989. ACM Press.

[97] David Spence and Tim Harris. XenoSearch: Distributed Resource Discovery in the XenoServer Open Platform. In *HPDC '03: Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing (HPDC'03)*, page 216, Washington, DC, USA, 2003. IEEE Computer Society.

[98] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 149–160, New York, NY, USA, 2001. ACM Press.

[99] V. S. Sunderam. PVM: A Framework for Parallel Distributed Computing. *Concurrency: Practice and Experience*, 2(4):315–339, 1990.

[100] Rob van Nieuwpoort, Thilo Kielmann, and Henri E. Bal. Satin: Efficient Parallel Divide-and-Conquer in Java. In *Euro-Par*, pages 690–699, 2000.

[101] B. Veeravalli, D. Ghose, and T. G. Robertazzi. Divisible Load Theory: A New Paradigm for Load Scheduling in Distributed Systems. *Cluster Computing*, 6(1), January 2003.

[102] Michael Voss and Rudolf Eigenmann. Dynamically Adaptive Parallel Programs. In *ISHPC '99: Proceedings of the Second International Symposium on High Performance Computing*, pages 109–120, London, UK, 1999. Springer-Verlag.

[103] Kevin Daniel Wayne. *Generalized Maximum Flow Algorithms.* PhD thesis, Cornell University, 1999.

[104] Ben Y. Zhao, John D. Kubiatowicz, and Anthony D. Joseph. Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing. Technical report, University of California at Berkeley, 2001.