2006

P2PCompute - A Peer-to-Peer Computing Model

Jayant Mishra
University of North Florida

P2PCOMPUTE – A PEER-TO-PEER COMPUTING MODEL

by

Jayant Mishra

A thesis submitted to the
Department of Computer and Information Sciences
in partial fulfillment of the requirements for the degree of

Master of Science in Computer and Information Sciences

UNIVERSITY OF NORTH FLORIDA
DEPARTMENT OF COMPUTER AND INFORMATION SCIENCES

December, 2006

The thesis "P2PCompute – A Peer-to-Peer Computing Model" submitted by Jayant Mishra in partial fulfillment of the requirements for the degree of Master of Science in Computer and Information Sciences has been

Approved by the thesis committee: Date

Signature Deleted

11/6/06

Sanjay Ahuja
Thesis Advisor and Committee Chairperson

Signature Deleted

11/7/06

Roger Eggen

Signature Deleted

11/6/06

Robert Roggio

Accepted for the Department of Computer and Information Sciences:

Signature Deleted

11/7/06

Judith L. Solano
Chairperson of the Department

Accepted for the College of Computing Sciences and Engineering:

Signature Deleted

11/9/06

Neal Coulter
Dean of the College

Accepted for the University:

Signature Deleted

13 NOVEMBER 2006

David Fenner
Dean of the Graduate School

ACKNOWLEDGEMENT

I wish to specially thank my spouse, Rinku, for her unwavering support and understanding during the many hours I dedicated to achieving this milestone. I also wish to thank my wonderful children, Milind and Tanisha, for bearing with me while I worked on this research. Thanks are also due to my parents and family who encouraged me to always dream higher and motivated me to get this degree.

This thesis would not have been conceptualized and completed without the direction provided by the thesis adviser, Dr. Sanjay Ahuja. His guidance and help are much appreciated. I would also like to thank Drs. Eggen and Roggio for agreeing to be on the thesis committee and providing tips and encouragement. It has been my privilege to work under these three distinguished UNF faculty members. I also wish to thank Jeff Bowen, UNF CIS Computer Applications Coordinator, for installing and maintaining the Tomcat servers used for the testing.

CONTENTS

FIGURES

TABLES

ABSTRACT

Peer-to-peer (P2P) networks consist of nodes which have both client and server capabilities and on which communication and data sharing is carried on directly between nodes, rather than being arbitrated by an intermediary node. The P2P architecture was popularized by file-sharing, one of the widely-used applications of the Internet. Many applications that are based on this architecture have been developed. It also provides an efficient platform to harness the computing power of a network of desktop computers. P2P computing power can help solve computationally complex problems that require powerful supercomputers. However, it has not been as widely used as the file-sharing P2P applications. Almost all of the current P2P computing applications are non-commercial endeavors. Users make their computing power available for these endeavors because they believe in the applications' objectives, for example, the SETI project analyzes radio telescope data in the quest for life in other parts of the universe.

This thesis proposes P2PCompute – a viable commercial model in the P2P computing field. It harnesses existing technologies – P2P, Java, the Internet and the UDDI registry, to enable distributed processing of tasks on multiple servers. It is well-suited to the heterogeneous environment on the Internet and has the potential to provide the spark that would lead to the development of more commercial P2P computing applications.

Chapter 1

INTRODUCTION

There has been an inexorable march in the computing world toward a more decentralized architectural approach. The earliest computer systems used centralized mainframe systems as the source of computational resources. These systems provided dummy client terminals for user interaction. Slowly, the trend moved toward providing more power to client terminals. This started the era of the client-server architecture and culminated in the advent of the Internet. The Internet itself spawned a host of changes in the computing world, one of them being the birth and widespread adoption of the peer-to-peer network architecture. This architecture takes the idea of distributed computing one step ahead and does away with the concept of specialized clients and servers. In an era of generalizations, this revolutionary architecture enables any node in the network to function as either a client or a server and, more importantly, change its role at will.

In the mainframe approach, almost everything is done by the central mainframe computer [Loo03]. This is a highly centralized approach with the mainframe system being the facilitator as well as the bottleneck for all tasks. In order to keep pace with technology and to constantly get better performance in the wake of changing processing requirements, the system needs to be upgraded periodically. This architecture is not well suited for scalability and has a single point-of-failure. Most early computer systems followed this model.

In the client-server computing paradigm, one or more clients and one or more servers, along with the underlying operating system and interprocess communication systems, form a composite system allowing distributed computation, analysis and presentation [Sinha92]. This architecture features one or more clients requesting services from a central server. The introduction of this architecture shifted some of the processing tasks to the client. It also enabled a move toward specializations with a server dedicated to a certain task. Clients were able to mix-and-match servers in order to complete a complex series of tasks, thus creating a richer set of functionality by daisy-chaining services. Through workload sharing, client/server systems can improve overall efficiency while reducing the budget for computing resources [Loo03]. Most of the existing systems follow this model.

A distributed system is one that looks to its users like an ordinary centralized system but runs on multiple, independent systems [Tanenbaum85]. The use of multiple systems to serve client requests is transparent to the user. Replication is used to achieve fault tolerance as well as provide better performance [Mullender96]. Both data and processes can be replicated thereby achieving greater performance through parallelism, increased reliability and availability, and higher fault tolerance [Soares92]. The use of distributed systems also allows incremental system growth by adding or replacing individual components [Schroeder93]. This enables the system to be more scalable.

Peer-to-peer (P2P) systems are distributed systems without any centralized control or hierarchical organization, where the software running at each node is equivalent in

functionality [Liben-Nowell02]. This architecture empowers the nodes joining such a network to be both servers and clients. Using this architecture, we can harvest the combined power of all the nodes in a network to perform a complex, computationally intensive task much beyond the capability of a single server. It does not require any major upgrade to the existing hardware resources of a network to bring this idea to fruition. In fact, with just a software upgrade, the existing network with its nodes and resources can be turned into a normal peer-to-peer system. File-sharing is a widely explored area of the peer-to-peer architecture, with successful applications based on it. The peer-to-peer architecture also provides an interesting solution for complex problems requiring powerful supercomputers to be solved by a network of desktop computers.

Enhancing the concept of the P2P architecture, the Power Server Computing Model empowers a client to use the computational power of many servers simultaneously [Loo03]. The client divides the task in separate sub-tasks and requests servers across the Internet to process each sub-task. The model uses Sun's Java 2 Platform Edition (J2EE) application server to provide a platform-independent environment for the tasks to run. This research takes the concept of the Power Server Model further by proposing the P2PCompute model. It improves upon the Power Server Model by addressing the issues preventing its widespread adoption and purposes solutions to resolve them. Chapter 2 analyzes different types of P2P systems in detail. Chapter 3 discusses the Power Server Model and identifies its strong points that need to be carried forward in the proposed model and flaws which need to be corrected. The P2PCompute model along with its various components and features are described in detail in Chapter 4. The implementation

done to validate and test the model is described in Chapter 5, while the experimental results are discussed in Chapter 6. Directions for future research and analysis are detailed in Chapter 7.

Chapter 2

PEER-TO-PEER SYSTEMS

Peer-to-Peer (P2P) systems are a type of distributed computing system designed for the sharing of computer resources (content, storage, CPU cycles, etc.) by direct exchange, rather than requiring the intermediation or support of a centralized server or authority [Androutsellis-Theotokis04]. The P2P architecture has seen a lot of interest in the recent past due to the popularity of file-sharing applications. Successful and widely-used applications based on P2P, like Napster, Gnutella, Freenet, and Kazaa have brought the P2P model out of the research field into the popular domain.

The P2P model is different from the traditional client-server model [Mishra04]. It has a decentralized architecture, thus each node is potentially equal in status to any other node in the network. This creates a lateral relationship among the nodes, rather than the traditional vertical relationship which gives the whole peer group tremendous processing power and storage space [Samtani02]. This architecture is more scalable since the addition of nodes provides more nodes with server capabilities, which increases performance and efficiency. However, the underlying architectural issues are much more complex than a traditional client-server model. Some of the challenging issues include:

1. Managing a heterogeneous mixture of nodes having multiple operating system platforms with different interfaces,

2. Managing the changing dynamics of the P2P network, with new nodes joining the network and old nodes leaving the network continuously and randomly, and

3. Managing the security policies of such a network, which is not inside a closed Local Area Network (LAN) within an organization, but on the public Internet.

Applications using the P2P model can broadly be divided into three categories – file-sharing, distributed processing and instant messaging [Damiani02]. Most of the P2P applications belong to the file-sharing category. As the name implies, this category enables users to share files, mostly MP3 and some shareware.

The computational power-sharing applications use the P2P model to share the computational power of the nodes in the network. These applications can enable a network of common workstations to perform tasks generally done by powerful supercomputers. SETI@home (Search for Extraterrestrial Intelligence) is an example of a well-known distributed processing P2P application. It uses millions of computers in homes and offices around the world to analyze radio signals from space. This approach, while complicated, delivers unprecedented computing power and has led to a unique public involvement in science [Anderson02].

P2P instant messaging applications involve sharing simple messages (either text or voice) or simple files using a messaging environment. Instant Messaging is no longer used merely to send messages, but has also become a major medium to stay in touch with

friends and share information [Rovers04]. AOL Instant Messenger, Yahoo! Messenger and MSN Messenger are some examples of instance messaging applications.

2.1 File Sharing Applications

Apart from the obvious use – sharing files – another use of these applications is content distribution through the P2P network. In fact, some anti-virus software producers are considering P2P networks as a convenient way to distribute virus signature updates. This technique will exploit the resiliency and aggregate bandwidth of P2P networks and avoid the overloading of central Web servers [Damiani02].

A typical P2P file-sharing client application works like this:

1. Search for nodes having the content needed by the user,
2. Make a peer-to-peer connection to that node,
3. Download the content, and
4. Disconnect.

The same node may also act as a server by spawning a daemon thread in the background to serve any requests from other peer nodes for contents it hosts.

One of the major challenges for the P2P file-sharing applications is to get the list of peer nodes having a particular content. The proposed solutions fall broadly under two major classifications – the "pure" P2P architecture, which does not have any central server; and

the "server-mediated" P2P architecture, which has a central server that maintain a registry of shared information and responds to queries for that information [Lui02]

The early version of Napster used a centralized directory server which maintained basic addressability and availability information about the user nodes and the meta-information about the shared files [Kant02]. This provided a simpler and faster search for the requested content. However, it had a single point-of-failure, thus enabling an easier shutdown when the courts decreed that Napster should stop operations due to copyright and piracy concerns.

In order to resolve this issue, some of the other P2P applications use a decentralized, "pure" P2P approach. Each node has a separate list of nodes with their addresses and available content. When a node joins the network for the first time, it should know the well-known address of at least one of the nodes already connected to the network. Once it connects to the network, it gets information from the already known node about all the nodes of which the latter is aware. It builds a list of such nodes. To get a file, it queries these nodes to find out which node has that file. Then it makes a direct TCP connection to the node that has the file. This way the file is transferred from one node to another. This approach is used by Gnutella and Freenet. Each node in the network also specifies a set of shared local storage areas that other nodes can search based on partial or full matches.

Once the address of the node which has the requested content is determined, most of the file-sharing applications connect to that address and retrieve the content. However,

Freenet introduced an innovative approach to take advantage of the current search. It caches the result so any subsequent requests for that content would be faster [Clarke01]. The requested content is returned back along the search route. Thus, each node in the request path caches the content. This technique improves performance and allows popular results to be cached at multiple nodes, enhancing redundancy.

Freenet also introduced other innovations to position itself as the application closest to the ideal distributed P2P application. It places emphasis on anonymity and makes it almost impossible to identify the source of content available on its network. Files are referred to in a location-independent manner, and are dynamically replicated in locations near requestors and deleted from locations where there is no request [Clarke01]. This has a kind of bubbling-up effect where files searched the most are available at multiple nodes, thus providing automatic replication for them. Files searched the least are rarely available at multiple nodes.

A major security challenge for such applications is to ensure that malicious data is not propagated through the network, masquerading as good data. The P2P architecture itself does not provide any protection against this issue. However, there are a number of solutions that can help in reducing malicious data [Oram01]. Restricting access using micropayments is one such solution. It envisages the peers having to offer something of value (money, CPU cycles, content, etc.) in order to be considered a part of the network. Another solution is to use reputation systems to rate popular or trustworthy resources and

nodes. Such systems are described in the literature [Kamvar03] [Damiani02] [Mengshu05] [Gupta03] [Walsh05].

2.2 Computational Power-Sharing Applications

Computational power-sharing applications are also known as distributed processing applications or P2P computing applications. Nodes that are part of this type of network make their idle CPU cycles available for use by others. This is mostly suited for a set of parallel computations known as "embarrassingly parallel" problems whose computational graph is disconnected [Fox94]. It is easier to decompose such problems into tasks which do not have any interdependency. Thus, each such task can be processed independently by any node in the network. Once the sub-tasks are completed, the results are returned back to the requestor node.

Most of the computational power-sharing applications require the users to download a small program to their computers. This enables communication with the client computer which needs the tasks performed. The tasks are automatically downloaded to the computer, performed and results communicated back to the requestor. This usually is done when the workstation is idle. Allowing these applications to run on one's workstation is voluntary. Participants believe in the cause supported by the application, e.g., SETI@home, so they allow it to run on their workstation. This simple model aggregates the power of common workstations to rival that of a supercomputer. In fact,

SETI@home can be considered to be the largest supercomputer in existence, having completed the largest computation ever performed [Korpela01].

2.3 Instant Messaging Applications

Instant messaging is a popular Internet technology that enables two or more users to communicate with each other using a client program. The key issues in architecting such applications are how to implement the user lookup and the message exchange mechanisms.

Most such applications, like Yahoo and MSN, use the centralized server approach, where user registration, lookup and message exchange are all done using dedicated servers. This approach is really a centralized one and not a true peer-to-peer model.

Other applications, like ICQ, enable message exchange using a peer-to-peer model, however, the user registration and lookup functionalities still use a central server. This improves the performance, since users do not want too much delay between writing and display of a line. These concerns are critical with the video conferencing and Voice over IP (VoIP) features being offered by a few of the instant messaging applications. However, this approach still suffers from a single point of failure where a malfunction will close the service, either by making it impossible to find clients or deliver messages.

Recently, there has been progress towards a true peer-to-peer system. [Lundgren03] describes the first fully distributed instant messaging system, named DIMA. The DIMA application runs on top of the Pastry peer-to-peer routing substrate [Rowstrom01]. It performs all necessary operations (join/leave, lookup, message exchange) without any centralized servers. Other applications take a different approach by using distributed hash tables to store the information. Lookups for keys are performed by routing queries through a series of nodes. Each node uses a local routing table to forward the query towards the node that is ultimately responsible for the key.

Chapter 3

POWER SERVER MODEL

3.1 Introduction

The P2P architecture enables file sharing, as well as computing power sharing. The Power Server Model introduced a new concept – a single client computer using the computing power of many servers on the Internet simultaneously [Loo03]. This model defines "power servers" to be computers connected to the Internet that provide CPU power to clients. Any computer on the Internet can be a power server by installing and running a J2EE application server. This model has the potential to extend the P2P computing model beyond the confines of selected projects to a wide variety of projects with possible business and financial implications. It utilizes existing tools and technologies, so minimal time and effort are required for deployment. It builds upon the computational power-sharing, or P2P computing model, and resolves key issues discouraging the widespread usage of P2P. In order to understand and critique the Power Server Model, it is important to analyze the issues surrounding the P2P computing model. This analysis is provided in the following section.

3.2 Drawbacks of the P2P Computing Model

The P2P computing model expands the processing power of a computer to encompass the collective power of the whole network. It has the potential to help a network do tasks

which previously were done only by expensive supercomputers. However, this model has some drawbacks preventing widespread commercial application to resolve different types of problems. The following are some of those issues [Loo03] :

- Security – Users sharing their computing power in the P2P computing model need to download a client program that runs on their computer. That program enables the use of their idle computing power to run the task for the client computer. However, downloading and running the client program on the Internet increases security risks to the computer where it runs. Any malicious code in such a program may be able to access local files and execute programs outside the control of the user. There is no security inherent in the P2P model to prevent such a program from doing this. This may make a lot of users apprehensive of running any client program from even a well-known organization, let alone provide unused CPU cycles for any organization on the Internet.

- Benefits – The users in such a model become part of the network only because they believe in and support the cause. There is no monetary or otherwise tangible benefit to the participants. The only benefit is to the cause and, of course, to the organizer. This prevents a lot of other organizations, especially commercial ones from donating their unused CPU cycles.

- Startup and upgrade issues – Typically, startup and upgrades are time-consuming, difficult and non-uniform for each such project. There is no automated process by

which a computer can announce its intention of donating CPU cycles for any client. It needs a manual effort to connect to the client computer and download the program with instructions on how to install and run it. It varies for each task and operating system. Similar work is performed for project upgrades. The client program needs to be downloaded again to get the upgrades. For users having multiple computers, the installation and maintenance require an inordinate amount of man hours, discouraging willing participants.

- Compatibility – The client programs that take advantage of the computing power of other computers are platform-specific. From the organizer's perspective, this represents a maintenance headache. Different versions need to be tracked and maintained to provide support and upgrades. It increases the maintenance cost as well as the complexity of the system. It also prohibits some power-sharing computers from joining a project that may not support their operating system. Maintenance also becomes a major issue for organizations having machines that run on different platforms.

3.3 How the Power Server Model Works

The Power Server Model was introduced to address the above issues and make the P2P computing model more acceptable. This model uses the Java 2 Enterprise Edition (J2EE) platform to resolve the issues. Since Java is platform independent, has a strong security

model (with security managers) and can be made to run tasks triggered from across the web, it is an ideal toolkit with which to start the next wave in the P2P arena.

The term "power server" in this model refers to the feature of providing CPU power to other users or client computers. As shown in Figure 1, a client computer divides a single, computation-intensive task into multiple small sub-tasks. Then it invokes a servlet on the power server, passing it the sub-task to be executed. The servlet in turn executes the sub-task and communicates the results to the client. The client aggregates all the results to get the consolidated result for the whole task.

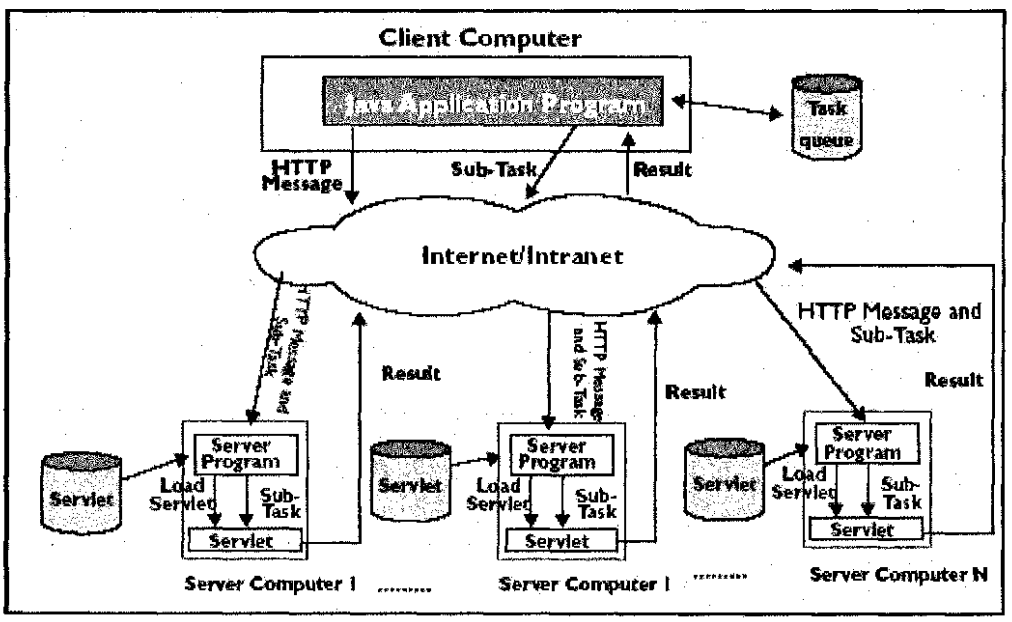

Figure 1: Power Server Model

3.4 Advantages of the Power Server Model

The power server model resolves the issues that are inherent in the P2P computing architecture.

- Security – This proposed model is more secure since it uses Java's well-known sandbox feature. It consists of security managers, which ensure the sub-tasks running on the server nodes do not impact any part of the server not exposed to the outside world. Since security is a part of Java and is automatically enforced, there is no additional package to be installed, or add-on cost that needs to be incurred. The simple Hyper Text Transfer Protocol (HTTP) is used to transfer the sub-tasks, as well as results back-and-forth between the clients and the servers.

- Business Benefits – There is a financial motivation as well for the organizations involved in this model, since any organization can host servers and provide such services for a fee. Clients also have the flexibility of choosing the most cost-effective server for running their tasks. This may be a viable business model that can be used by companies to earn profits.

- Startup and upgrade issues – Startup is easy since it just involves installing any J2EE-compliant server. Many good server software packages are available and most of them are freeware or shareware [Hunter01]. The software is used to run a standard servlet which has the capability of executing tasks requested by other

clients. The other advantage is the same servlet can be used for multiple projects. The bytecodes for the task are downloaded on the fly from the client site. Thus, there is no need for any server-side upgrade, if the task itself changes. The client should transmit the latest task when it connects to the server.

- Compatibility – The software can be executed on any J2EE-compliant application server. Due to the platform-independence of Java, the project is not tied to any particular platform. In fact, if a client has transmitted its tasks to more than one server, it may be possible for one task to be processed on a Sun server, whereas another could be processed on a Linux server, yet another on the Windows platform and so on. The client is only concerned about the results it receives. The cost of maintenance and the complexity is minimized for the clients since they do not need to maintain different versions of the code for different platforms.

3.5 Issues with the Power Server Model

There are a four primary issues associated with the Power Server Model :

- Finding Power Servers – One problem of this model is the difficulty in finding the power servers. A client has to know the IP address of any server before a connection is made. To resolve this issue, a separate infrastructure needs to be created and used to facilitate discovery of the power servers. A centralized coordinator server used to store the IP addresses of all power servers willing to

provide computing power can be added to the system [Loo03]. Any computer owner who wants to donate computer power to the network must register with the coordinator and provide some basic information. However, the clients have to know the IP address of the coordinator, in order to query it to get the list of power servers. Also, in case the primary coordinator is down, they need to have a backup coordinator in place and know its IP address, in order to get the list of power servers. This solution has the drawback of relying on the coordinator to store the IP addresses of all the nodes. However, once a node has the address information of other nodes in the network, it does not need the coordinator. From then on, all communication is between the peer nodes.

- Static Allocation of Tasks – The clients do not have any way of querying and getting the list of power servers. So, tasks are allocated statically to the same set of servers every time. The clients need to be informed about any changes to the servers. Changes may include a new server being added or a server going out of commission. There is no provision for such tasks in the Power Server Model.

- Power Server Failure – This model does not take into account the failure of the power servers. So, the clients have no way of finding out how many power servers from the static list it has are active. Thus, a client is forced to send its tasks to the servers it has in the static list and then wait for it to either timeout or get an error back. In case of an error, it has to repeat this with another server, until it gets an active server that responds successfully and processes its tasks.

- No Power Server Classification Mechanism – This is particularly significant for clients that have tasks with specific processing requirements. Such clients do not have any way of knowing which servers support the task requirements. For example, a client may need all its tasks to be processed within a particular time-frame on a fast processing server with no cost constraint. Alternatively, for another client the cost may be of paramount consideration. The Power Server Model does not provide any way to determine such information.

Chapter 4

P2PCOMPUTE

4.1 Introduction

This thesis proposes P2PCompute – a peer-to-peer computing model. The proposed model builds upon the Power Server Model, by enhancing it and resolving the issues associated with it. It takes advantage of the peer-to-peer architectural model to enable workstations on the Internet to share computing power. It uses the well-known concept of a Universal Description, Discovery, and Integration (UDDI) registry to set up a new peer-to-peer infrastructure. Using this infrastructure, clients can reach out to server organizations that provide computing power services. The model provides features to keep track of the loads on different servers dynamically and makes it easier for clients to access this information on demand. The information provided to the clients details the capabilities of each server and how much load the servers are handling currently. This enables the clients to choose servers on the fly, discarding any inactive servers, as well as those which do not match the requirements for the task at hand. These features have the potential to utilize the hitherto untapped power of the P2P computing architecture.

4.2 P2PCompute Design

The P2PCompute model uses existing Java technology and the Internet to create a powerful peer-to-peer mechanism for sharing computing power with the world. Figure 2 shows the architecture of the model proposed in this research.

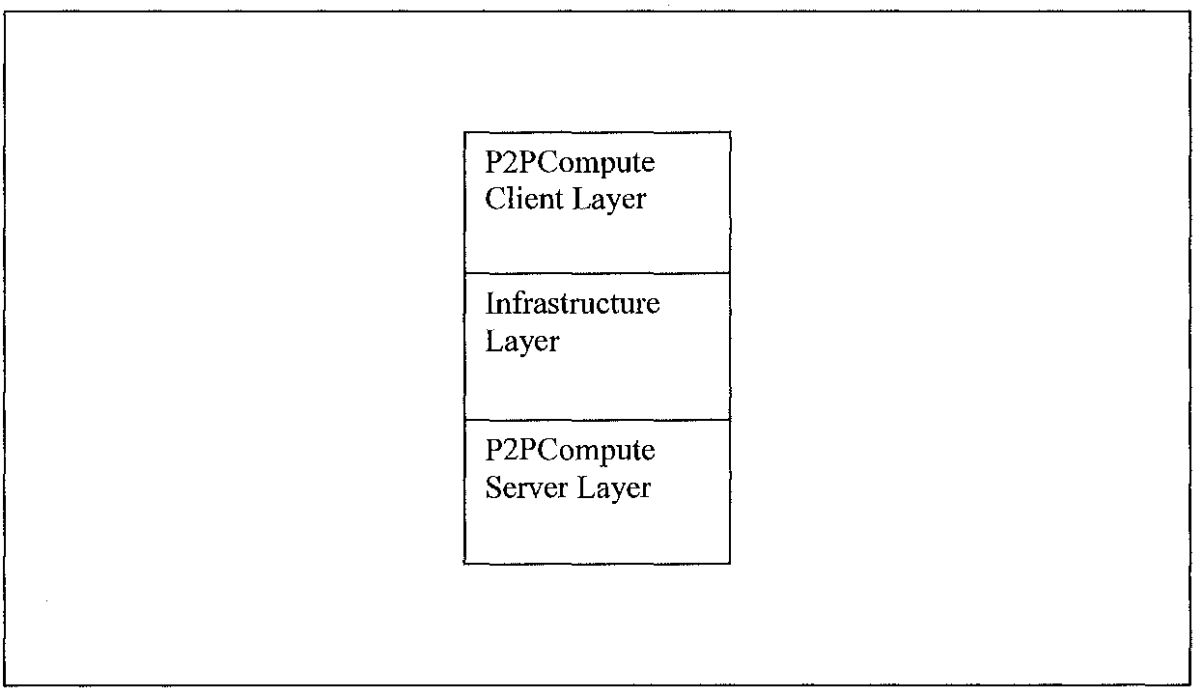

Figure 2: P2PCompute Architecture

As shown in Figure 2, there are essentially three layers a node in the P2PCompute network may have – Client, Infrastructure and Server. The client needs to go through the infrastructure layer to get to the server layer. Thus, the infrastructure layer is really a connectivity layer. Generally, a node may only have one layer, but it is conceivable to

have a node playing more than one role since it is a peer-to-peer architecture. A layer is chosen keeping in mind the specific task the node needs to do. For example, a node used just as a client needs just the client layer, whereas a node that functions both as a client and server has both the client and server layers. Each layer needs specific software packages to be installed on the nodes. The model provides enough flexibility that any node can be transformed from one type to another with just the addition of the new layer on that node.

Figure 3 shows a detailed view of the P2PCompute model. All interactions between the three architecture layers are through well-defined interfaces on the Internet. The client-side nodes get the list of servers from the infrastructure components of this model. Then they connect to the server-side nodes and transmit the bytecodes comprising the task to be processed. The server nodes execute the task and transmit the results back to the client. The following sub-sections analyze the functions and roles of each component of this model.

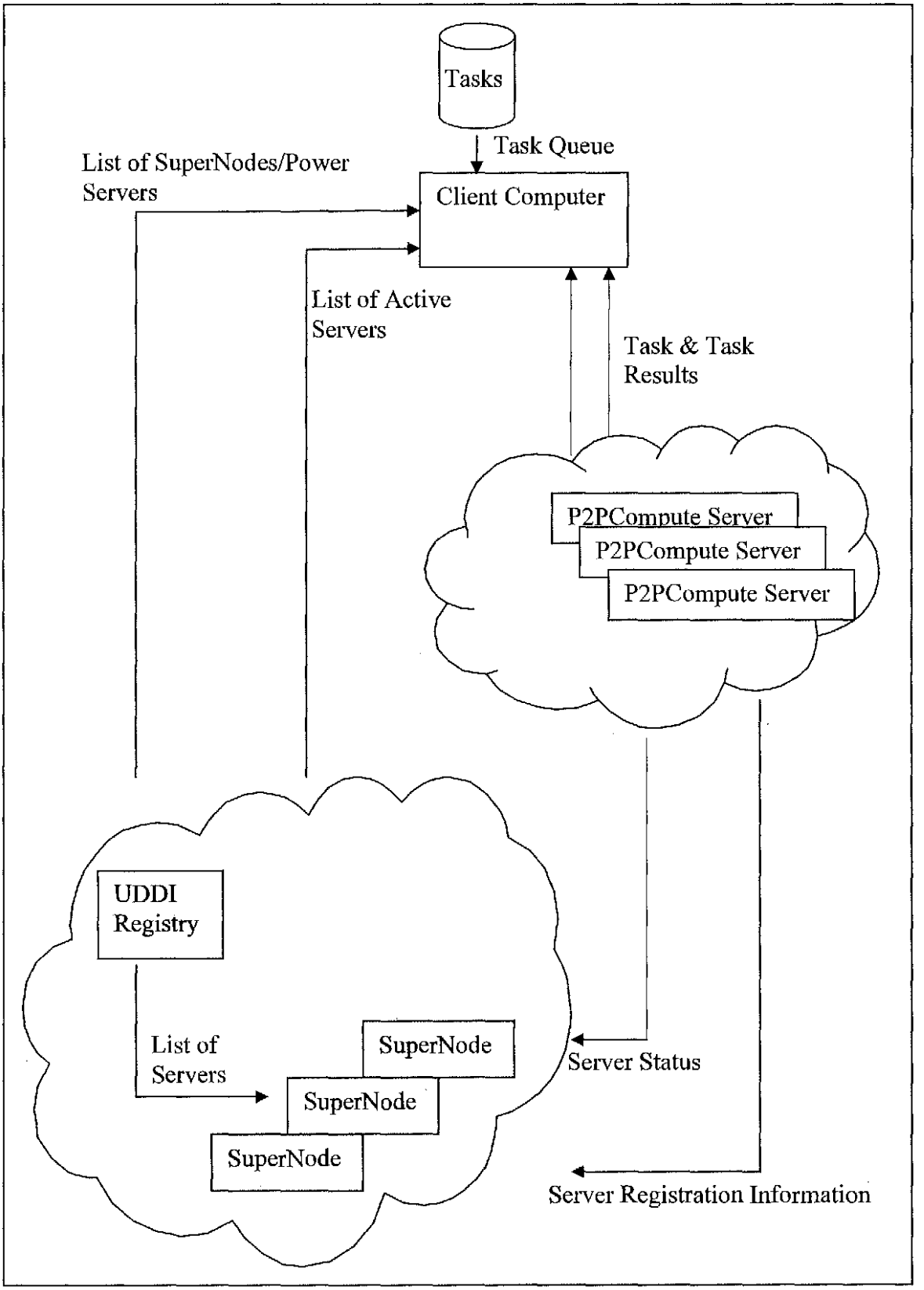

Figure 3: The P2PCompute Model

4.2.1 P2PCompute Infrastructure

The primary purpose of the P2PCompute Infrastructure layer is to provide a computing infrastructure to support the P2PCompute model. It matches clients needing computing power with servers willing to share computing power over the Internet. From a client's perspective, the function of the P2PCompute Infrastructure layer is to provide a uniform mechanism for any client on the Internet to find P2PCompute servers matching its requirements. From a server's perspective, its function is to keep track of active servers and have accurate and up-to-date information on them to feed to clients. In order to do that, this model uses UDDI registries and introduces the concept of supernodes. The following sections discuss both in detail.

4.2.1.1 UDDI Registry

UDDI provides a public registry to discover businesses and their services [Dogac02]. The UDDI registry servers function as both a white pages business directory as well as a technical specifications library. This enables clients to find organizations offering computing power over the Internet. The technical specifications library feature helps clients obtain details about the services offered. These details may include the technical capabilities of those servers, as well as other non-technical information like the financial costs of utilizing the servers. Some of these specifications are mostly static data that rarely change during the course of the life of a server, like the number of CPUs on the server. Some other specifications can change due to software or hardware upgrades, like

the maximum number of servlet threads may be increased due to an increase in the memory of the servers. In case of such upgrades, the servers may need to update their registration information in the registry. Following are some of the technical specifications recommended in this model:

- The URL to connect to
- The processor speed
- The number of CPUs on the server
- The maximum memory on the server
- The server's operating system platform
- The maximum number of servlet threads allowed on the server

4.2.1.2 Supernode

A supernode acts as a conduit between the registry and the servers. It first queries the registry to get a list of all registered servers which offer computing power on the Internet. Then it queries each server continuously to get their updated status. Any server which does not respond is marked inactive. Each active server returns current information about the server and updated status information, including the current load on the server and the current utilization of CPU and memory. It also reduces the number of idle servlet threads in the application server that are available for processing more tasks. Since this information is dynamic and subject to change, the supernodes query the servers frequently. The frequency should be determined by the supernode administrator, but it

should be finely balanced so it does not put extra load on the servers, as they have to respond to each such query in addition to executing the client task. Querying should be frequent enough, however, so the supernode does not have stale, outdated information about the server. Not having the correct information defeats the very purpose of supernodes. These capabilities enable the clients to make an informed decision on which servers to mix-and-match for their specific needs, thus enabling customization of the servers by the clients.

4.2.2 P2PCompute Server

The P2PCompute Server layer is essentially comprised of P2P nodes capable of running as servers. In order for them to be considered part of the P2PCompute model, they need to register with the P2PCompute Infrastructure layer as computing power providers. This will enable the supernodes and the clients to find them easily and connect to them to transmit the bytecodes to execute. The servers should run any Java 2 Enterprise Edition (J2EE) application server. These application servers provide the ability to do the tasks expected of them as part of the P2PCompute Server layer. From the client's perspective, the servers in this layer listen to and accept any connection requests from the clients. Once connected, they receive the Java bytecodes sent by the client on the Internet. These bytecodes are instantiated into a class and loaded into the application server's JVM. The servers then execute the task and transmit the results back to the client.

From the P2PCompute Infrastructure layer's perspective, first and foremost, the servers in the Server layer need to register with the Infrastructure layer. The registration information must include the URL(s) which can be used to connect to the servers, as well as useful static information. The servers should also be able to respond to status requests from the supernode to indicate if they are active and convey status information about themselves. Such status information enables the P2PCompute Infrastructure layer to provide a better overview of servers that may meet the needs of the tasks must be executed at a particular point in time. Table 1 describes the fields a server is expected to pass back to the supernode, in the status response.

Information	Description	Possible Values
Status code	This field provides information about the current status of the server. The server may be busy, active or inactive. For example, it may send an inactive code even though it is up and on the network, when it is not taking any task requests, or it is undergoing maintenance.	BUSY, ACTIVE, INACTIVE
CPU Usage	The current CPU usage, in percent	From 0% to 100%
Memory Usage	The current memory utilization, in percent	From 0% to 100%
Threads Actively Used	The current number of threads actively working on tasks, in percent	From 0% to 100%
Application Server Name	The name of the J2EE application server that is used by the server	Tomcat, JBoss, Websphere, Weblogic, etc.
Application Server Version	The J2EE application server version number	
Maximum Memory Allocated	The maximum memory size that the application server is allowed to use	
Maximum Threads	The maximum number of request processing threads that can be used by the server. This determines the maximum number of simultaneous requests that can be handled.	

Table 1: Status Information Fields in the P2PCompute Server Response Message

4.2.3 P2PCompute Client

The P2PCompute Client layer of the P2PCompute model is comprised of any workstation on the Internet which is capable of transmitting its tasks as bytecodes to another workstation or server on the Internet. The client may have a list of tasks to be performed in either its database or on the client itself. It examines each task one-by-one, dividing them further into smaller tasks, if necessary. Using the P2PCompute Infrastructure layer, it finds the appropriate servers on which to execute each task. It is recommended that a task not be allocated to any server with capacity utilization above 90%. The tasks are then uploaded to the servers as bytecodes. Once the tasks are finished, the client gets the results back and does any post-processing, if needed. Thus, the P2PCompute model simulates having multiple processors even though the client may have a single, common local processor.

The challenge in this layer is to find the server on which a task is to be executed. This is accomplished using the P2PCompute Infrastructure layer, more specifically the UDDI registry and the supernodes. The P2PCompute model proposes two alternative approaches. The first approach using just the UDDI registry servers is simpler; however, it does not provide dynamic up-to-date information about the servers, so the client may not necessarily choose the appropriate server for its needs. The second approach uses both the UDDI registry servers and the supernodes. It is more complex, however, it has the advantage of providing current server status to enable the client to make a more informed decision. The following section discusses both approaches.

4.3 Proposed Approaches to Find Servers

4.3.1 Using the UDDI Registry Directly

This is a simple method for getting a list of servers. The client sends a request to any well-known UDDI registry server. This request queries the UDDI registry for any servers which offer computing power services. The registry responds with the information it has on such servers, such as the server URL, which can be used to connect directly to the server to get the task executed. The P2PCompute model also proposes having other attributes / specifications of the server in the registry. This may include the maximum number of threads the server's servlet pool has available, the hardware specifications, and the amount of memory the server has. Such information will be of immense help in enabling the client to choose servers according to its needs. For example, a client whose task may need a lot of memory will discard those servers which have less memory, instead of trying to execute the task on those servers and then running out of memory later on.

The disadvantage of using the UDDI registry is that the registry does not include up-to-date information on the server, e.g., the current load, or the current CPU and memory utilization. Such factors are important if the client has time or memory constraints. This lack of information also prevents this approach from providing for automatic load-balancing. As a consequence, servers with excellent specifications may get overloaded

with too many requests and have a hard time catching up. On the other hand, other less powerful servers may sit idle with few tasks coming their way.

4.3.2 Using the UDDI Registry and Supernodes

This approach enables clients to examine the changing dynamics of servers and helps them choose servers better suited for executing their tasks. The client still queries the UDDI registry, but this time to get the list of supernodes. Supernodes provide the client with dynamic information on the servers. Upon querying the supernode, the client gets the list of all active power servers, which in itself is an improvement over just using the UDDI registry. This approach also provides an updated information on the active servers. This includes information on factors such as the number of servlet threads currently executing tasks for other clients, the amount of memory being consumed, and the CPU utilization. To compile this information, each supernode querics all the servers that are in the UDDI registry at specified intervals to find out how busy they are at a particular point in time. The responses back from the servers are stored in the updated list the supernode maintains on the active servers. This provides a valuable service to the clients by giving them the data to choose the best possible list of servers for their needs. It also provides an automatic load balancing feature, since it prevents over loading a single server.

The disadvantage of this approach is its complexity and the extra call to the supernode. In eases where the client does not have any time, performance, or memory constraints, it adds an unnecessary step. However, in other cases, it is worth the effort, since it ensures

the client will not waste its time waiting on a server that is not responding, or that does not currently have resources needed for executing the task.

4.4 Features

4.4.1 Load Balancing

Load balancing is one of the key features of the P2PCompute model. It is accomplished using the supernodes in the P2PCompute infrastructure layer. The supernodes continuously poll all the servers in the UDDI registry to get their current status. The status information includes the server's updated usage statistics on the CPU, memory, and thread usage. These statistics are expressed in percentages instead of absolute terms. Percentages give a better idea of the current server capacity utilization in relation to its total capacity. Since the supernodes continuously poll the servers, this information is subject to change at any given time. When a client queries the supernodes, it gets back the list of active servers with the updated status information. Then the client divides its task equally among these active servers, skipping any servers with capacity utilization over 90%. For example, if a task involves processing 1GB of data and there are 16 active servers, all under 90% utilization, each would be tasked with processing 64MB of data.

4.4.2 Scalability

Scalability, in its most general form, is defined as the ability of a solution to a problem to work when the size of the problem increases [Rana00]. The following three dimensions need to be analyzed in order to discuss scalability in the context of this research:

- An increase in the number of servers,
- An increase in the number of clients requesting their individual tasks to be processed, and
- An increase in the size of the data that is part of the tasks the servers are processing.

Increasing the number of servers makes more processing power available on the network. Each resulting task becomes simpler to process, because it can be distributed among more servers and, therefore, becomes less intensive computationally and have a smaller data-size. Adding new servers does, however, add a little more load to the supernodes, which need to add these additional servers to their polling list.

Increasing the number of clients or the amount of data to be processed in the P2PCompute model does put a strain on the existing resources in the network. It increases the number of tasks the servers need to process, as well as increasing requests to the UDDI registry and the supernode. The existing clients also are impacted; since they need to compete with the new clients for servers. It may also cause delays in processing

the tasks, if, for example, the clients have more tasks than the servers can handle. This would result in most of the servers being at more than 90% capacity and cause the clients to wait for the servers to come back to normal capacity levels.

However, with the P2PCompute model, the increase in clients and data is spread out across all the servers, so the performance deteriorates at a much slower rate than if the task were done on the client side itself. In fact, the deterioration rate is minimal with an increase in the number of servers. So, if this model is widely used with many available servers, theoretically the increase in data or clients may not be significant. Also, from a business perspective, this represents an opportunity for organizations to increase their business by increasing the number of servers they make available.

4.4.3 Redundancy and Error Handling

From the server's perspective, the P2PCompute model provides for redundancy, by having multiple servers available for clients. In case of failure of a server, there are other servers that can be used by the clients. The supernodes periodically query each server to get their updated status. If no response is received, the server is flagged as inactive. If a server fails in the middle of processing a task, the client connection times out waiting for results, allowing the client to use another server. P2PCompute does not provide any commit functionality to save intermediate states while a server is processing a task. Therefore, the incomplete task needs to be processed again on another server.

The design of the P2PCompute infrastructure layer ensures both the UDDI registry and the supernodes are redundant systems. There are multiple well-known UDDI registry servers on the internet. For example, both IBM and Microsoft provide free access to their public UDDI registry servers. As far as the supernodes are concerned, the UDDI registry contains a list of supernodes. When a client queries the registry to get the list of supernodes, all the supernodes in the list are returned. This ensures that, if the first supernode is not available, the client can try accessing other supernodes in the list.

4.4.4 Platform Independence

The model's use of Java and the J2EE application server promotes platform-independence. The clients, supernodes, UDDI registry and servers can run on any platform, as long as they can communicate with each other via TCP. In fact, the UDDI registry and the supernodes do not even need to run Java. UDDI registries already exist which use diverse technologies, like J2EE or .NET to serve requests. The supernodes just need to run any application server (e.g., J2EE, .NET, Coldfusion). The only dependency in the P2PCompute model is that the task the client needs to run should be compiled to Java bytecodes. This promotes platform-independence, but not language-independence.

4.4.5 Incentives

When a server registers with the UDDI registry, the registration information provided may include the financial remuneration expected for executing a task. Servers may use

different types of pricing structures. For example, servers may post on the registry their computing power rental charges based on CPU time required, or CPU cycles or memory size used. This introduces an additional burden on the server to keep track of the cost-determining factor (e.g., the CPU time required) when running the task for a client. Once the task is processed, the task results are sent to the client, along with a report detailing the cost of processing the task. For audit purposes, the cost should also be logged on the server side.

There may be two ways in which payments can be handled. Upon establishing the connection with a client, the server should ask for the client's account number with the server. If the client responds with a valid account number, then the server should process the task, return the task results to the client along with an invoice for executing the task, and add the charge to the client's account number. The client should have the option of paying a monthly consolidated invoice. This approach may be used for trusted clients that have an on-going business relationship with the server. Accounts should be set up for such clients on the server. A different way may be used for one-time or unknown clients. If a client does not have an account with the server, it may respond with a credit card number. Upon completion of the task, the credit card should be charged the amount invoiced to the client.

Chapter 5

IMPLEMENTATION

5.1 Server Implementation

The server implementation runs on Tomcat, a J2EE application server. It uses a servlet TaskHandler to service all the tasks. This class listens to any connect requests from the clients and accepts it. The client connects using the URL connection mechanism; however, other connection mechanisms, like SOAP or Web Services may be used. Once the connection is established, the servlet downloads the bytecodes of the task to be executed from the client. It then loads the bytecodes as a class in the servlet's JVM to execute. A custom class loader that extends the ClassLoader class was developed during this implementation to load the bytecodes coming from the wire into the JVM. It first loads an object deserializer, which is used to unmarshall bytecodes into an object. The task defined is executed and the results serialized back onto the output stream that is connected to the client.

The task itself implements a well-known interface named Task with a single method process() that has to be defined by the class that implements this interface. The Task class is available on both the server and the client side. This way the server knows which method to call to execute the task. Clients implement the process() method in the Task class. This method contains the task the server needs to execute. Once the server deserializes and loads the object it got from the client, it proceeds to execute the process()

method. This in turn runs all the processing the client needed. Once the task is processed, the results are communicated to the client via the URL connection mechanism.

5.2 P2PCompute Infrastructure Implementation

The supernode implementation runs as a daemon thread on a Tomcat application server. However, it may as well be implemented as a standalone Java daemon class. It is implemented as two separate classes. The first one, ServerMonitor interfaces with the UDDI registry and the servers, while the second one, SuperNode responds to client requests. The ServerMonitor first queries the UDDI registry, which is set up on a Tomcat-jwsdp server to get a list of all servers and their static information. That information contains the URL of the servers. It then connects to that URL and issues a status request to the server. The server responds with the status information, including the current server utilization (e.g., CPU, memory and thread utilization). All this information is recorded on the supernode. When the clients request active servers from the supernode, the SuperNode class works to get the list from the ServerMonitor class which is given to the client. The client chooses the servers that are the best fit for the tasks to be processed.

A class to manage the UDDI registry entries is also implemented to ensure the full P2PCompute model is given a thorough test. This class can add, delete and query UDDI registry entries on the jwsdp server. It is also implemented as a servlet thread on the application server running the UDDI registry. The supernodes and also some clients that

do not use supernodes may call this servlet thread to get details on the servers available for processing their tasks.

5.3 Client Implementation

The client implementation consists of a few classes. The main class, TaskRequestor, is used to spawn multiple threads, depending on the parameters passed. Each thread is an instance of the RequestHandler class and does all the work to get the task processed. The TaskRequestor class connects to the UDDI registry to get the list of supernodes. Then it chooses one of the supernodes to get the list of all active servers, their URLs, and the maximum number of threads it can support. Based on the number of threads, it divides its big task, especially one requiring a lot of data-crunching, into multiple tasks among all the threads. The task to be processed is defined in the TaskImpl class, which implements the Task interface, thus the process method has all the details needed by the server to execute the task.

Each RequestHandler thread connects to the server it is assigned to and uploads the task. Once the task is processed, the results are returned to the thread. The thread may then return the results to the TaskRequestor to do any post-processing, such as aggregating the results.

Chapter 6

RESULTS AND ANALYSIS

6.1 Test Configuration

The configuration for the two machines used in this research is described in Table 2. To test the P2PCompute functionality, a total of 6 clients were used to simulate multiple requests to the servers. For the servers, 4 machines were used to test functionality like load balancing and dynamic task allocation. To ensure reliability on the supernode side, 2 supernodes – a primary and a secondary one – were used in the test case. Both the servers and supernodes were multithreading capable.

	Configuration of Supernodes and Servers	
Processor	Dual-CPU Pentium III 450MHz	Quad-CPU Hyper threaded (emulates 8-CPU) Pentium 4 Xeon 1.5GHz
Memory	256 MB	8 GB
Hard Disk	18GB RAID-5 Array	263GB RAID-5 Array
Operating System	Linux SMP	Linux SMP
Software	Tomcat J2EE Application Server, Oracle Client, PHP	Tomcat J2EE Application Server, Oracle Client, PHP
Services	Apache, Tomcat, MySQL	Beowulf cluster manager, Apache, Tomcat, MySQL, Oracle

Table 2: Test Configuration

6.2 Comparison with the Power Server Model

Since the P2PCompute model is closely tied to the Power Server model, a detailed test and analysis was done to see how it compares to that model. The following sections discuss the results of the comparison.

6.2.1 Performance Comparison

A performance comparison test was performed to demonstrate the benefits of the P2PCompute model. Two servers were used for this test, with one of them having a total of 2 threads active and the other 150 threads active at the point in time the clients were trying to connect to them. The test involved comparing the processing times taken by the Power Server Model and two implementations of the P2PCompute model. The first implementation used the UDDI registry to get the server information and then connect to the servers. The second implementation gets the list of supernodes from the UDDI registry and then queries the supernodes to get the list of active servers with status and other information. As discussed in the previous chapters, the second approach is the recommended approach. The task data size was steadily increased to get more readings and to analyze performance deterioration when the data size is increased from 64 KB to 256 KB. The resulting chart in Figure 4 sheds some light on the advantages realized by the supernode concept in this model and the division of the task into multiple threads in the same server.

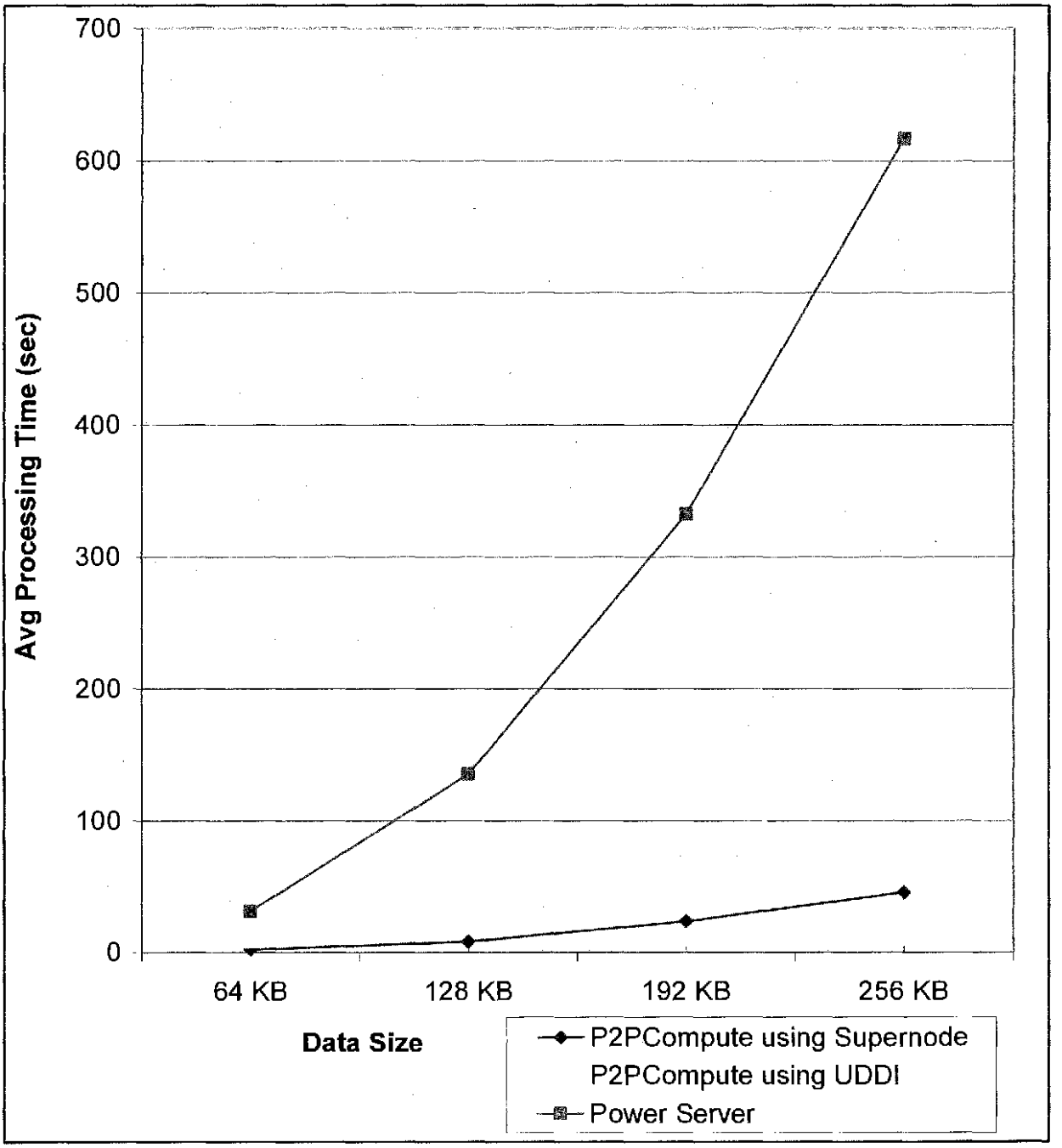

Figure 4: Performance Comparison of P2PCompute and the Power Server Model

The results show the following:

- Comparison of P2PCompute and the Power Server Model: Both the P2PCompute models performed better than the Power Server Model. The P2PCompute model using the UDDI registry directly resulted in a decrease in processing time by an average of 76.32%, whereas the other P2PCompute model decreased the processing time by an average of 93.04%. These substantial performance gains may be explained by the fact that both the P2PCompute models take advantage of multiple P2PCompute servers. The Power Server Model does not have a way to determine how many power servers are active on the Internet at a particular time. The number of power servers has to be configured in advance on the Power Server side. The Power Server had knowledge of a single server before the test, so it used that. The P2PCompute model which queried the UDDI registry knew both the servers, since an extra server registered during this time. The P2PCompute model which uses the supernode is even more intelligent, since it knew there was more than one available thread on each server. Hence, the performance gain with the P2PCompute model using supernodes was more than that of the P2PCompute UDDI model.

- Comparison of both the P2PCompute implementations: The P2PCompute implementation with the supernode lookup performed better by an average of 70.5% than the implementation with the direct UDDI lookup. The details from the supernode helped the client, since it showed the server's number of active threads.

Thus, the client could use multiple server threads on both the servers to process its request. However, the direct UDDI lookup implementation could not know the number of active threads and hence used just a single thread of hoth the servers.

- Performance of P2PCompute implementation with data size increase: With the P2PCompute implementation, the processing time increases at a much slower rate than with the Power Server Model implementation. This is due to the fact that the increase in data size is divided equally among all the server threads. Thus, the increase in the processing time of each individual thread is not as much for the P2PCompute implementation. This accounts for a lower performance deterioration rate. The average performance deterioration was calculated from the increase in processing time, when the data size is increased from 64KB to 256KB. Experimental data showed the deterioration to be 134% for the P2PCompute Supernode implementation, and 131% for the P2PCompute UDDI implementation, whereas for the Power Server Model it was 140%.

6.2.2 Scalability Comparison

A scalability comparison was done to determine how well the P2PCompute model scales in comparison with the Power Server model. Increasing the number of P2PCompute servers improved the performance in the P2PCompute model. However, increasing the number of servers with the Power Server model does not improve performance. This is explained by the fact that the Power Server model does not determine the power servers

dynamically and requires code or configuration changes on the client side to use the new servers. In comparison, the P2PCompute model can adapt to the changing dynamics of the P2PCompute servers, due to constant polling and querying being performed by the supernodes. Also, when any server is brought down or opts to go out of the network, it would not get any more connection requests. In comparison, in the Power Server model, inactive servers would still get connection requests, since the clients do not have any way of knowing the servers are no longer in service.

To perform this test, data was divided into equal, constant size units to be processed by each server. The number of servers was increased from 2 to 8 in increments of 2. Figure 5 shows the processing time decreases in the P2PCompute model as the number of servers are increased, since there are more servers available to distribute the tasks. In contrast, the Power Server model did not realize the addition of more servers, hence continued using the same single server it started with. Therefore, the processing time remained the same with the Power Server model.

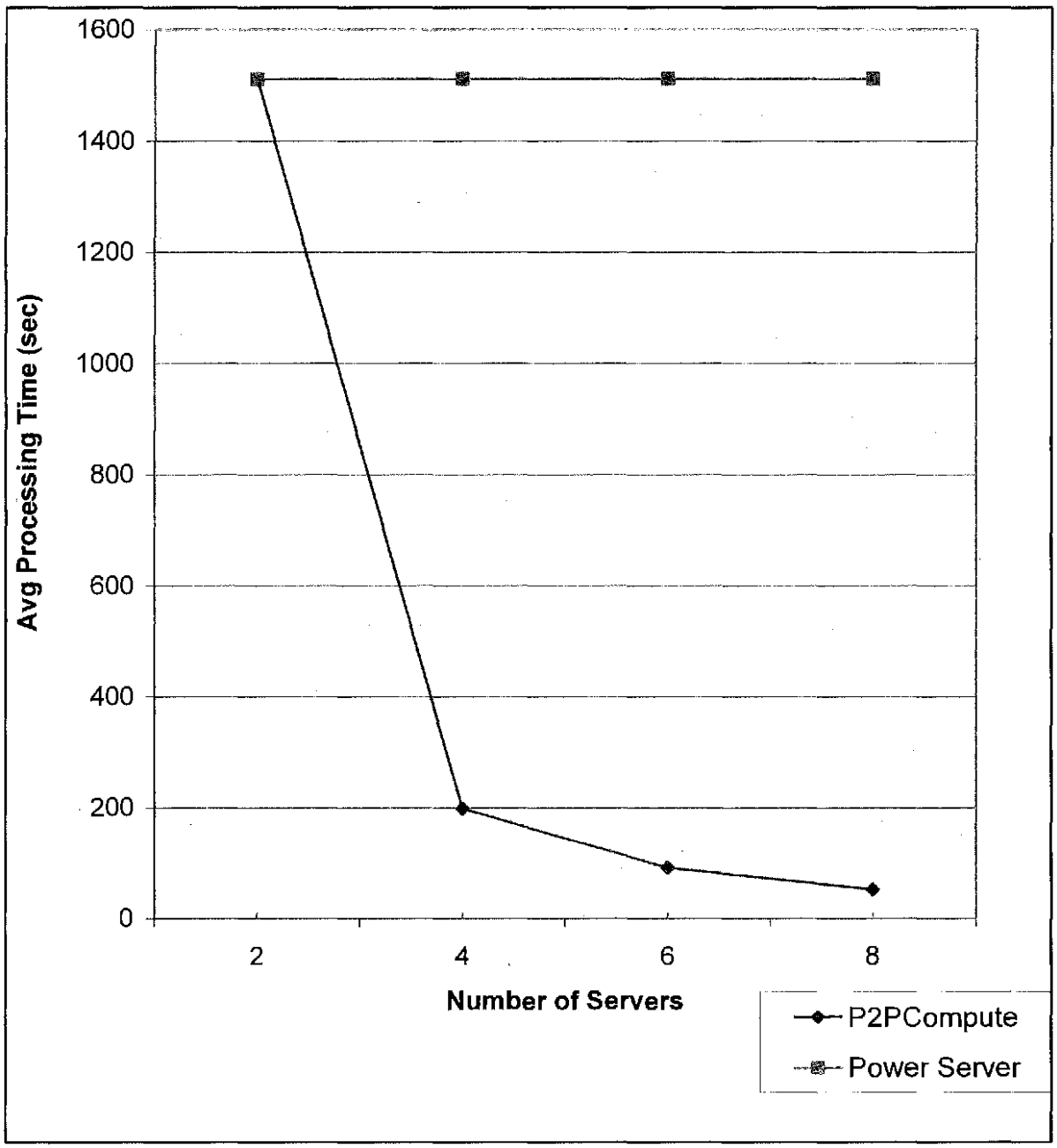

Figure 5: Scalability Comparison of P2PCompute and the Power Server Model

6.3 Standard Algorithmic Tests

A detailed test was carried out to benchmark the P2PCompute model vis-à-vis standard algorithms. Three algorithms were chosen to simulate the complexities corresponding to O (n), O (n log n) and O (n^2).

The client implemented the task interface by defining the task to be one of the standard algorithms. The processing time taken by the P2PCompute model was calculated from the time the client transmitted the data to the time it received the processed results back from the server. For comparison purposes, the actual times taken, by a stand-alone Java program, to run these standard algorithms were also recorded. The amount of data to be sorted by each algorithm was varied from 512KB to 1024 KB. To ensure a fair comparison, only one server was used in the P2PCompute model. Of course, the P2PCompute model took some more time due to the network part of the model. Time is required for the client to transmit the data to the server, and then receive the processed data back. However, the graphs resulting from these tests provided a good idea of how this model compares with graphs resulting from just running standard algorithms.

6.3.1 Sequential Search – O(n)

The sequential search algorithm O(n) was the simplest algorithm used to test the behavior of the P2PCompute model. This algorithm iterates over a list of data elements, comparing each such element to the desired element. The result is a count of the times the desired

element occurs in the list. As a corollary, from this count, it can be inferred whether the desired element occurs in the list.

Figure 6 shows the results from running the sequential search algorithm in the P2PCompute environment compared with results from running the algorithm in a stand-alone environment. The processing time for the P2PCompute model increases steadily as the size of the input data increases. The P2PCompute model took more time compared to the standard stand-alone implementation due to the network overhead. The rate of increase of the processing time with the P2PCompute model approximates a linear line, which is the expected behavior for O(n) algorithms.

Compared to the stand-alone environment, the P2PCompute model does not do as well as the stand-alone process on two counts. First, the processing time itself is more in the P2PCompute environment. Second, the rate of increase in the processing time is more in this environment compared to the stand-alone environment. These issues can be explained by the extra time taken in the P2PCompute model to transmit the increasing amount of data (from 512KB to 1MB) from the client to the server and to get the processed results back from the server. This shows the P2PCompute model is not well suited for algorithms with an O(n) complexity.

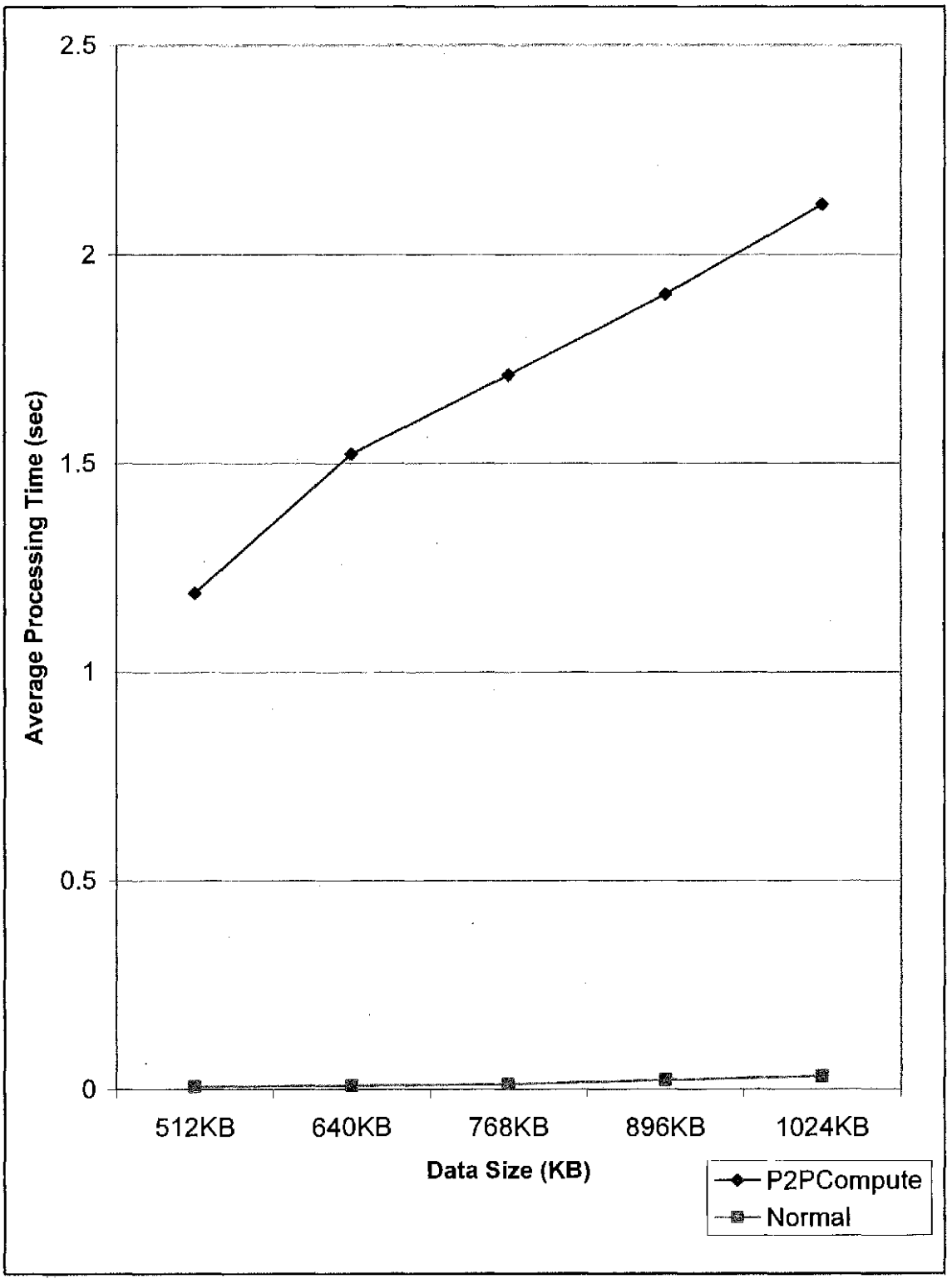

Figure 6: Sequential Search Algorithm Results

6.3.2 Quick Sort – O(n log n)

The quick sort is one of the more sophisticated sort algorithms which are quicker, yet more difficult to code, due its use of the divide-and-conquer concept and a massively recursive mechanism. The algorithm itself is similar to a merge sort, however, it differs in the way the input list is split into multiple sub-lists. Each sub-list is sorted recursively and merged to get the sorted list. Based on the inverted sorting taxonomy proposed [Merritt85], quick sort uses the hard split/easy join technique, instead of the easy split/hard join technique used by a merge sort.

The following four steps comprise the heart of this recursive algorithm [Nguyen01]:

1. If there is one or less element in the array to be sorted, return immediately.

2. Choose any of the elements from the array to serve as a "pivot" point. The first element in the array was used in this test. This is what is generally used.

3. Split the array into two parts - one with elements larger than the pivot and the other with elements smaller than the pivot. This would rearrange the array in such a way that all elements to the left of the pivot are less than it and all elements to the right of the pivot are greater or equal to it.

4. Recursively repeat the algorithm for both halves of the original array till the first step returns the element. This would sort the list.

The efficiency of the algorithm is impacted by which element is chosen as the pivot point. If the list is already sorted, it yields the worst performance of the quick sort, with the complexity being $O(n^2)$. Otherwise, the quick sort should have an algorithmic complexity of O(n log n).

Figure 7 shows the processing times taken when using the quick sort in the P2PCompute model environment and when run stand-alone.

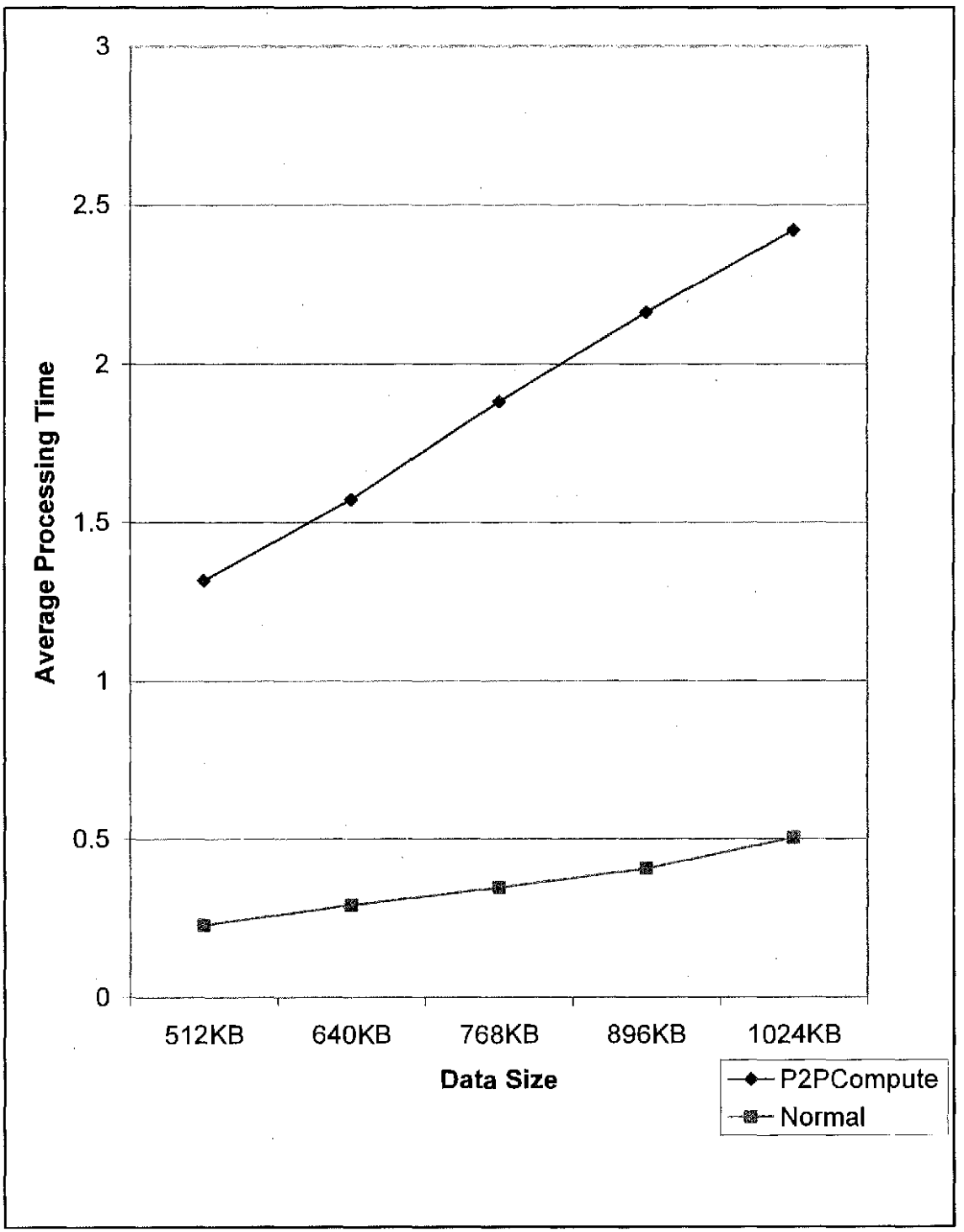

Figure 7: Quick Sort Algorithm Results

Again, similar to the results from the O(n) comparison, the processing time increases when running under the P2PCompute model, due to the extra network time taken to transmit data back and forth. The rate of increase in processing time with the increase in the data size conforms for the most part to the standard quick sort rate of processing time increase. Therefore, even this comparison also shows the P2PCompute model is not a performance efficient model for running tasks having O(n log n) complexities.

6.3.3 Bubble Sort – $O(n^2)$

The bubble sort is one of the oldest sorting techniques in use, though it is one of the slowest. It is a comparative sort, because it determines which interchanges to make by comparing two elements at a time [Martin71]. The bubble sort works by comparing each item in the list with the item next to it and swapping them, if required. The algorithm repeats this process until it makes a pass all the way through the list without swapping any items. This causes larger values to "sink" to the end of the list, while smaller values "bubble" towards the beginning of the list, hence the name.

This sorting algorithm is generally considered to be the most inefficient sorting algorithm. It is almost never used except for cases where there are a small number of elements in the list and coding simplicity is preferred over performance. Figure 8 shows the results from tests done to compare this algorithm running under the P2PCompute model as compared to running it as a stand-alone process.

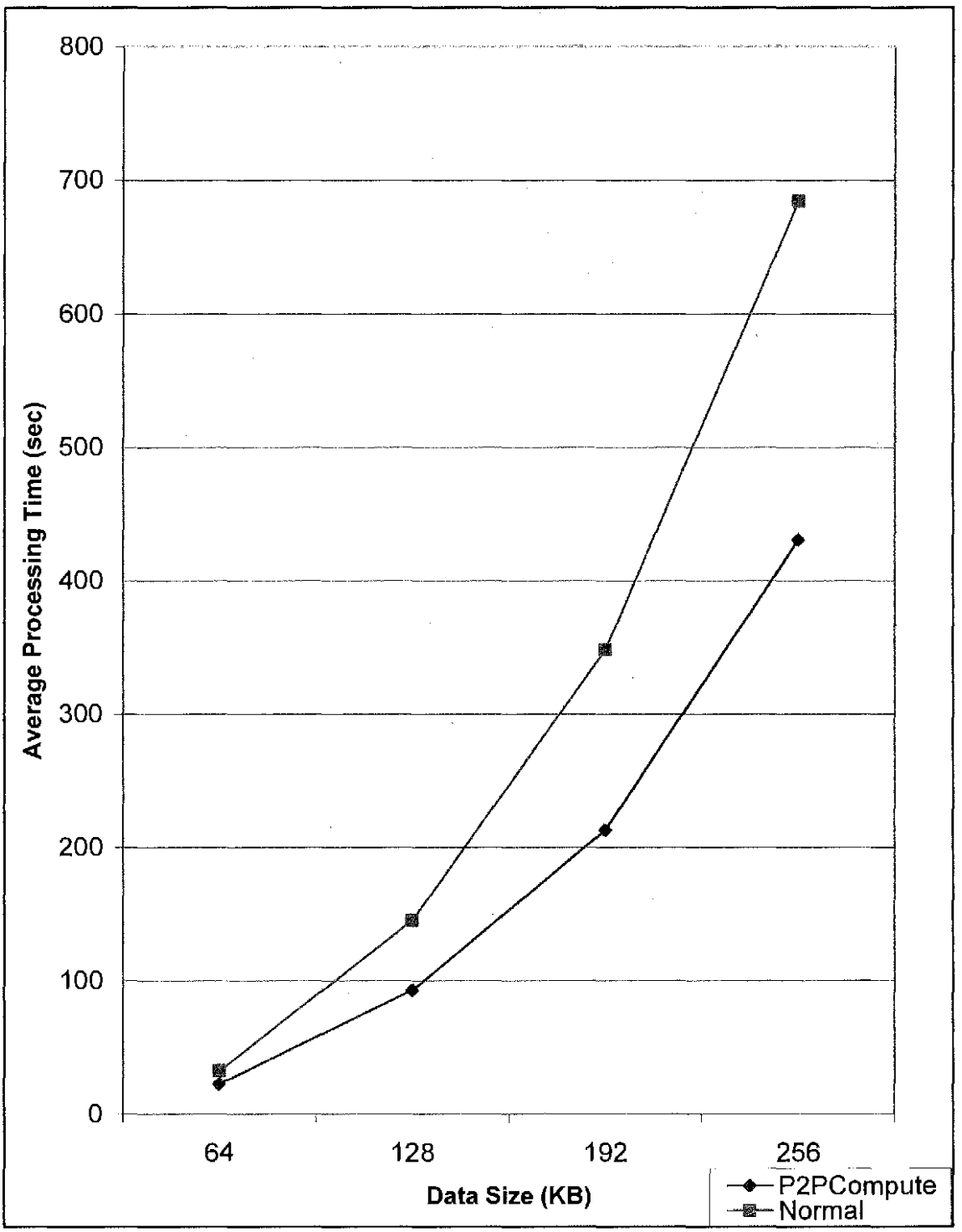

Figure 8: Bubble Sort Algorithm Results

The graph looks similar in terms of the gradient and shape of the graph for both implementations. Also the difference in both these implementations is not that as much in the case of O(n) and O(n log n) complexity implementations. This is due to the fact that the total CPU time is significantly much more than the network time. This shows how it would be better for clients not having enough computing power for such complex tasks as those with $O(n^2)$ complexity, to use the P2PCompute model. It should be noted that in these tests, the algorithm was run as a single process and on two servers with the P2PCompute model. In reality, the clients may connect to more than 2 servers, divide the task into subtasks and then assign the subtasks to the multiple servers. This would create opportunities for performance gains, since it would simulate running the tasks on multiple processors. In such a situation it is expected the performance gain would be greater in the P2PCompute model for algorithms with $O(n^2)$ complexity, as compared to the stand-alone server.

6.4 Load Balancing

The P2PCompute architecture provides the ability to balance the client requests among multiple servers. This feature was tested by comparing two P2PCompute implementations – one of which had the load balancing feature turned off and the other had the feature turned on. The number of registered and active servers on the P2PCompute network was kept constant at 4 servers, whereas the client load was increased steadily from 64KB to 256KB in increments of 64KB. For each data size, four readings of the processing time were recorded to get a fair average. This was repeated

twice, once with the load balancing feature turned on and then turned off. Figure 9 shows the exponential rise in processing time with an increase in the client data size for the implementation with no load balancing. In contrast, the other implementation does much better and also scales nicely with increase in the client data size. This shows the benefit offered by the load balancing feature of the P2PCompute model. However, the model itself does not force the clients to use this feature. They may choose not to use the supernodes and instead use any of the servers registered in the UDDI registry. This shows the flexibility and the customization feature offered by this model.

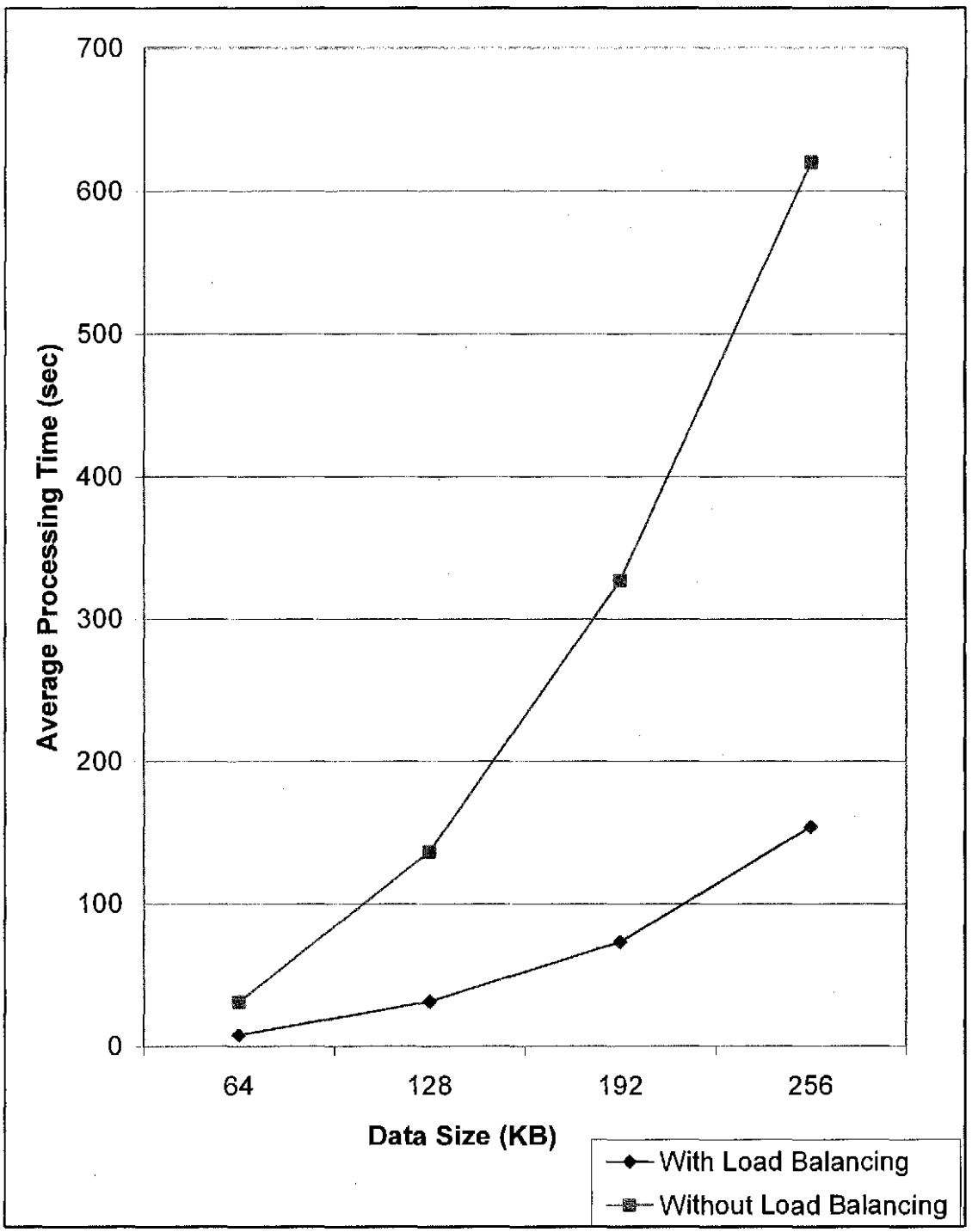

Figure 9: Load Balancing Results

6.5 Supernode Validation

The addition of the supernode concept in the P2PCompute model introduces an additional layer to the model. It enables clients to realize performance gains by taking advantage of the dynamic structure of the Internet, where new servers are continuously being added. It also enhances the productivity of the servers by not overloading them with multiple requests. The clients benefit by being directed to servers that are not busy. In order to validate the gains realized by the addition of the supernode concept to the P2PCompute model, a performance analysis was done to compare the performance with and without the supernodes. The P2PCompute model took advantage of the supernode to determine 2 servers on the Internet, whereas the Power Server model just used the single server that was statically allocated to it. Figure 10 shows the processing time for the P2PCompute model was much less than the Power Server model. The increase in processing time with increase in data size was also less for the P2PCompute model than for the Power Server model.

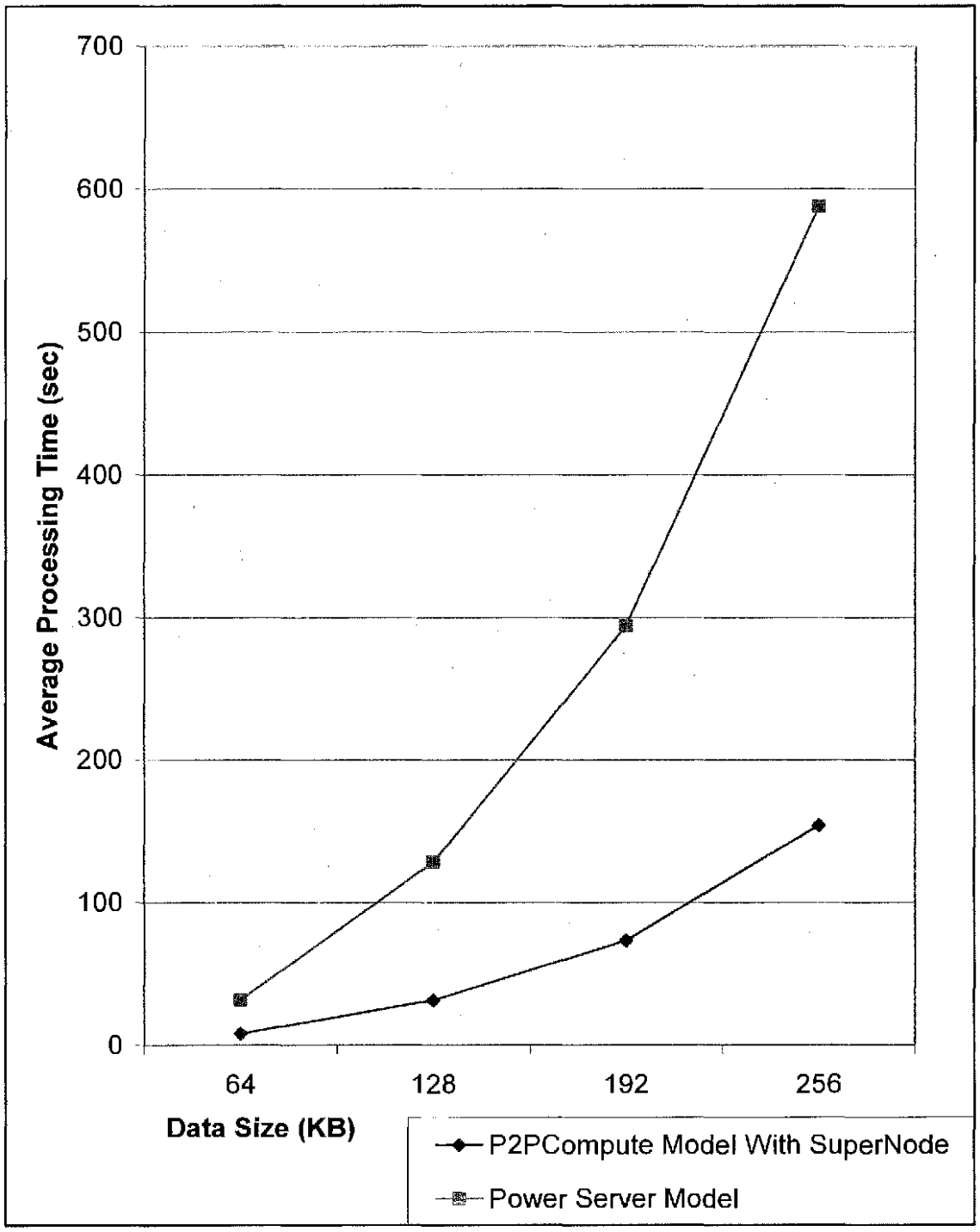

Figure 10: Supernode Validation Results

Chapter 7

CONCLUSIONS AND FUTURE RESEARCH

7.1 Conclusions

The P2PCompute model provides a solid framework that can be used for commercial implementations. It provides an elegant way to use the idle power of countless computers on the Internet, without the use of any new hardware infrastructure. This untapped computing power could be utilized to perform tasks that are currently performed only by supercomputers. This model uses existing technologies like the UDDI registry, Java, J2EE application server and the Internet. The P2PCompute model enables users to focus on defining the task they need to do, instead of worrying about having the computing power to support it. It also has the potential to create a new business model of organizations selling computing power on the Internet.

Based on experimental results, the P2PCompute model is well-suited for large, computationally intensive tasks, which can be divided into subtasks to be processed in parallel. For example, this research demonstrated that tasks which have the complexity of $O(n^2)$ perform well in the P2PCompute model, especially when they run on multiple servers. However, tasks which are simpler and smaller do not perform as well. Examples of such tasks are the sequential search and the quick sort which have the respective complexities of O(n) and O(n logn). Their processing time is overshadowed by the additional time added by the P2PCompute model. This additional time includes the time

it takes to get the list of supernodes from the UDDI registry, get the list of servers from the supernodes, split the task and connect to each server to get the task processed.

The P2PCompute model also encourages load balancing by dividing the client tasks, distributing them to multiple servers, and keeping track of the current utilization of the servers. Experimental results also demonstrated the scalability of this model as addition of more servers adds more computing power to this model, as referred in Section 6.2.2. The performance gains from this model are also notable. As it keeps track of the servers, the model enables clients to take advantage of the addition of more servers. This is in addition to the performance gain it already gives due to the parallel processing paradigm it embraces.

7.2 Future Research

7.2.1 Security and Privacy Issues

This research used Java, which provides a secure sandbox security mechanism for the tasks to run. So, from a technical and micro level, the security is tight. However, security issues still merit consideration on the macro level. For example, to get financial remuneration, a P2PCompute server needs to prove it did the work for a particular client. It is imperative servers be able to authenticate and authorize clients. This is necessary to ensure clients do not masquerade as another client that already has a valid account and dupe the server into executing their tasks.

The server should also have a mechanism to unload the client task classes once the task has finished. This would provide reassurance to the clients their copyrighted task classes would not be reverse-engineered by dubious servers and their intellectual property stolen.

Even though the P2PCompute implementation in this work was done using the HTTP protocol, there is no protocol-specific setting in the P2PCompute model itself. Thus, other protocols may be used in the communication between clients and servers. For example, some servers may try to distinguish themselves from the others by offering a more secure, but higher priced HTTPS / SSL connection. This is worthy of further research and analysis, since it potentially would provide more security to client tasks, the input data, and the results.

7.2.2 Redundancy and Error Handling Issues

Redundancy and error handling are implemented in the P2PCompute model. However, if a server crashes while executing a task, the client would get a timeout. Should the client try another server to process the interrupted task, the next server would have to start over, thus losing any progress made by the original server. Future research to enhance the model is recommended so the next server can start from the point of the crash of the first server. This may involve having multiple commits on the server side into the supernodes, to ensure the next server can pick up from where the first one left.

7.2.3 Remuneration / Financial Aspect

A log / audit mechanism should be devised to enable both the supernodes and P2PCompute servers to share the financial rewards. Presently, the P2PCompute model has a proposed a mechanism to compensate the servers, however, it does not provide for compensation of the supernodes. The supernodes provide valuable service to the clients and servers. They act as intermediaries between them and need some financial incentive for their work. One approach for compensating the supernodes might involve the servers giving a small percentage of their earnings from the tasks they are performing to the supernode that referred the client to them. For this, the P2PCompute model needs to be changed to ensure the clients pass the referral node name to the server when they are connected. A second approach might involve the clients compensating the supernodes for each query they request. These two approaches represent possible future research.

REFERENCES

[Anderson02]
Anderson, D. et al., "SETI@home: An Experiment in Public-Resource Computing," Communications of the ACM, Volume 45, Issue 11, November 2002, pp. 56-61.

[Androutsellis-Theotokis04]
Androutsellis-Theotokis, S. and D. Spinellis, "A Survey of Peer-to-Peer Content Distribution Technologies," ACM Computing Surveys, Vol. 36, No. 4, December 2004, pp. 335–371.

[Clarke01]
Clarke, I. et al., "Freenet: A Distributed Anonymous Information Storage and Retrieval System," International Workshop on Designing Privacy Enhancing Technologies: Design Issues in Anonymity and Unobservability, Berkeley, CA, USA, January 2001, pp. 46-66.

[Damiani02]
Damiani, E. et al., "A Reputation Based Approach for Choosing Reliable Resources in Peer-to-Peer Networks," Proceedings of the 9th ACM Conference on Computer and Communications Security, 2002, pp. 207-216.

[Dogac02]
Dogac, A. et al., "An ebXML Infrastructure Implementation Through UDDI Registries and RosettaNet PIPs," International Conference on Management of Data Archive, Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, 2002, pp. 512-523.

[Fox94]
Fox, G., R. Williams, and P. Messina, "Parallel Computing Works," Morgan Kaufmann Publishers, 1994.

[Gupta03]
Gupta, M., P. Judge, and M. Ammar, "A Reputation System for Peer-to-Peer Networks," International Workshop on Network and Operating System Support for Digital Audio and Video Archive, Proceedings of the 13th International Workshop on Network and Operating Systems Support for Digital Audio and Video, Monterey, CA, USA, 2003, pp. 144-152.

[Hunter01]
Hunter, J. and W. Crawford, "Java Servlet Programming," O'Reilly, Sebastopol, CA, 2001.

[Kamvar03]
Kamvar, S., M. Schlosser, and H. Garcia-Molina, "The EigenTrust Algorithm for Reputation Management in P2P Networks," Proceedings of the Twelfth International Conference on World Wide Web, 2003, pp. 640-651.

[Kant02]
Kant, K., R. Iyer, and V. Tewari, "A Framework for Classifying Peer-to-Peer Technologies," Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID'02), May 2002, pp. 368-375.

[Korpela01]
Korpela, E. et al., "SETI@home: Massively Distributed Computing for SETI," Computing in Science and Engineering, Volume 3, Issue 1, January 2001, pp. 78-83.

[Liben-Nowell02]
Liben-Nowell, D., H. Balakrishnan, and D. Karger, "Analysis of the Evolution of Peer-to-Peer Systems," Proceedings of the Twenty-first Annual Symposium on Principles of Distributed Computing, July 2002, pp. 233-242.

[Loo03]
Loo, A., "The Future of Peer-to-Peer Computing," Communications of the ACM, Volume 46, Issue 9, September 2003, pp. 57-61.

[Lui02]
Lui, S. and S. Kwok, "Interoperability of Peer-to-Peer File Sharing Protocols," ACM SIGecom Exchanges, Volume 3, Issue 3, 2002, pp. 25-33.

[Lundgren03]
Lundgren, H., R. Gold, E. Nordström, and M. Wiggberg, "A Distributed Instant Messaging Architecture based on the Pastry PeerToPeer Routing Substrate," Swedish National Computer Networking Workshop (SNCNW2003), September 2003.

[Martin71]
Martin, W. A., "Sorting," ACM Computing Surveys (CSUR), Volume 3, Issue 4, December 1971, pp. 147-174.

[Mengshu05]
Mengshu, H. et al., "A Trust Model of P2P System Based on Confirmation Theory," ACM SIGOPS Operating Systems Review Archive, Volume 39, Issue 1, January 2005, pp. 56-62.

[Merritt85]
Merritt, S., "An Inverted Taxonomy of Sorting Algorithms," Communications of the ACM, Volume 28, Issue 1, January 1985, pp. 96-99.

[Mishra04]
Mishra, J. and S. Ahuja, "A Survey of the State of the Art in Peer-to-Peer Computing," 3rd International Conference on Communications, Internet and Information Technology (CIIT 04), November 2004.

[Mullender96]
Mullender, S., "Distributed Operating Systems," ACM Computing Surveys, Volume 28, Issue 1, March 1996, pp. 225-227.

[Nguyen01]
Nguyen, D. and S. Wong, "Design Patterns for Sorting," Proceedings of the Thirty-Second SIGCSE Technical Symposium on Computer Science Education, 2001, pp. 263-267.

[Oram01]
Oram, A. et al., "Peer-to-Peer: Harnessing the Power of Disruptive Technologies," O'Reilly & Associates, March 2001, Chapter 16.

[Rana00]
Rana, O. and K. Stout, "What is Scalability in Multi-Agent Systems," International Conference on Autonomous Agents, Proceedings of the Fourth International Conference on Autonomous Agents, 2000, pp. 56-63.

[Rovers04]
Rovers, A. and H. Van Essen, "HIM: A Framework for Haptic Instant Messaging," Conference on Human Factors in Computing Systems, CHI '04 Extended Abstracts on Human Factors in Computing Systems, 2004, pp. 1313-1316.

[Rowstrom01]
Rowstrom, A. and P. Druschel, "Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems," Proceedings of Middleware'01, November 2001.

[Samtani02]
Samtani, G. and D. Sadhwani, "Web Services and Peer-to-Peer Computing," Tect Ltd., July 2002.

[Sinha92]
Sinha, A., "Client-Server Computing," Communications of the ACM, Volume 35, Issue 7, 1992, pp. 77-98.

[Schroeder93]
Schroeder, M. D., "A State-of-the-Art Distributed System: Computing with BOB," In Distributed Systems, 2nd ed. S. J. Mullender, Ed. ACM Press, New York, pp. 1–16.

[Soares92]
Soares, P.G., "Distributed Systems - Programming and Management: On Remote Procedure Call," Proceedings of the 1992 Conference of the Centre for Advanced Studies on Collaborative Research, Volume 2, pp. 215-267.

[Tanenbaum85]
Tanenbaum, A. and R. Renesse, "Distributed Operating Systems," ACM Computing Surveys (CSUR), Volume 17, Issue 4, December 1985, pp. 419-470.

[Walsh05]
Walsh, K. and E. Sirer, "Fighting Peer-to-Peer SPAM and Decoys with Object Reputation," Proceeding of the 2005 ACM SIGCOMM Workshop on Economics of Peer-to-Peer Systems, 2005, pp. 138-143.

APPENDIX A: P2PCOMPUTE CODE LISTINGS

```
/********************************* Begin Task class ***************************************/
package edu.unf.p2p.common;

public interface Task {
        public void process();
        public void postProcess();
}
/********************************* End Task class *****************************************/


/********************************* Begin RequestHandler class *****************************/
package edu.unf.p2p.client;

import java.io.*;
import java.util.*;
import java.net.*;

import javax.xml.registry.*;
import javax.xml.registry.infomodel.*;

import edu.unf.p2p.common.Task;
import edu.unf.p2p.util.Log;

/**
 * This class requests the server to run the task
 */
public class RequestHandler extends Thread {
        private String serverURL;
        private int dataLen;
        private int id;
        private boolean done = false;
        private long timeTaken = 0L;
        private static final String MY_NAME = "RequestHandler";

        public RequestHandler(String serverURL, int dataLen, int id) {
                this.serverURL = serverURL;
                this.dataLen = dataLen;
                this.id = id;
        }

        public void run() {
                Log.logDebug(MY_NAME, getId() + " processing req, URL: " + serverURL);
                long startTime = -1;

                try {
                        URL url = new URL(serverURL + "?cmd=");
                        URLConnection con = url.openConnection();
                        con.setDoInput(true);
                        con.setDoOutput(true);
                        con.setRequestProperty("Content-Type", "application/octet-stream;");

                        int numBytes = -1;
                        byte[] byteArr = new byte[30 * 1024];

                        OutputStream out = con.getOutputStream();

                        Task task = new TaskImpl(dataLen);
                        Class taskClass = task.getClass();
                        ClassLoader loader = taskClass.getClassLoader();
                        InputStream inStream = loader
                                        .getResourceAsStream("edu/unf/p2p/client/TaskImpl.class");
```

```
                        numBytes = inStream.available();
                        startTime = System.currentTimeMillis();

                        ObjectOutputStream numOut = new ObjectOutputStream(out);
                        numOut.writeInt(numBytes);
                        numOut.flush();

                        BufferedInputStream taskIn = new BufferedInputStream(inStream);
                        BufferedOutputStream bufOut = new BufferedOutputStream(out);
                        while ((numBytes = taskIn.read(byteArr, 0, 30 * 1024)) != -1)
                                bufOut.write(byteArr, 0, numBytes);
                        bufOut.flush();

                        // Write object with ObjectOutputStream
                        ObjectOutputStream objOut = new ObjectOutputStream(out);
                        objOut.writeObject(task);

                        objOut.flush();
                        objOut.close();
                        bufOut.close();
                        out.close();
                        taskIn.close();
                        inStream.close();

                        InputStream in = con.getInputStream();
                        ObjectInputStream objIn = new ObjectInputStream(in);
                        task = (TaskImpl) objIn.readObject();
                        Log.logDebug(MY_NAME, getId() + " Got task back from server");

                        objIn.close();
                        numOut.close();
                        in.close();
                } catch (Exception e) {
                        e.printStackTrace();
                        System.out.println("got exception: " + e.getMessage());
                        return;
                }

                done = true;
                long endTime = System.currentTimeMillis();
                timeTaken = endTime - startTime;
                Log.logDebug(MY_NAME, getId() + " Processed request in " + timeTaken
                                + " msec");
        }

        public boolean isDone() {
                return done;
        }

        public String getId() {
                return "Thread #" + id;
        }

        public long getTimeTaken() {
                return timeTaken;
        }
}
/******************************** End RequestHandler class ******************************/


/******************************** Begin TaskImpl class **********************************/
package edu.unf.p2p.client;

import java.io.*;

import edu.unf.p2p.common.Task;
import edu.unf.p2p.util.Log;
```

```
/*
 * This class implements the Task to be performed at the server
 *
 */
public class TaskImpl implements Task, Serializable {
        public static final boolean DEBUG = false;
        public static final int RANDOM_MULTIPLY_FACTOR = 100000;
        public static final int ARRAY_LEN = 8000;

        private int[] numArr;

        public static void main(String[] args) {
                int dataLen = Integer.parseInt(args[0]);
                TaskImpl task = new TaskImpl(dataLen);
                long startTime = System.currentTimeMillis();
                task.process();
                long endTime = System.currentTimeMillis();
                System.out.println("Time to service the request: "
                                + (endTime - startTime) + " msec");
        }

        public TaskImpl(int len) {
                numArr = createArr(len);
        }

        public static int[] createArr(int len) {
                int[] retArr = new int[len];
                if (DEBUG)
                        System.out.print("Array: ");
                for (int i = 0; i < len; ++i) {
                        retArr[i] = (int) (Math.random() * RANDOM_MULTIPLY_FACTOR);
                        if (DEBUG)
                                System.out.print(retArr[i] + " ");
                }
                if (DEBUG)
                        System.out.println();
                return retArr;
        }

        public void process() {
                bubbleSort(numArr, numArr.length);
                //q_sort(numArr, 0, numArr.length-1);
                //search(numArr, (int) (Math.random() * RANDOM_MULTIPLY_FACTOR));
        }

        public void postProcess() {
                System.out.println("Sorted Array: ");
                for (int i = 0; i < numArr.length; ++i) {
                        System.out.print(numArr[i] + " ");
                }
                System.out.println();
        }

        public static void bubbleSort(int[] numbers, int array_size) {
                int i, j;

                for (i = (array_size - 1); i >= 0; i--) {
                        for (j = 1; j <= i; j++) {
                                if (numbers[j - 1] > numbers[j]) {
                                        int temp = numbers[j - 1];
                                        numbers[j - 1] = numbers[j];
                                        numbers[j] = temp;
                                }
                        }
                }
        }

        public static void q_sort(int[] numbers, int left, int right) {
                int pivot, l_hold, r_hold;
```

```
                l_hold = left;
                r_hold = right;
                pivot = numbers[left];
                while (left < right) {
                        while ((numbers[right] >= pivot) && (left < right))
                                right--;
                        if (left != right) {
                                numbers[left] = numbers[right];
                                left++;
                        }
                        while ((numbers[left] <= pivot) && (left < right))
                                left++;
                        if (left != right) {
                                numbers[right] = numbers[left];
                                right--;
                        }
                }
                numbers[left] = pivot;
                pivot = left;
                left = l_hold;
                right = r_hold;
                if (left < pivot)
                        q_sort(numbers, left, pivot - 1);
                if (right > pivot)
                        q_sort(numbers, pivot + 1, right);
        }

        public static int search(int[] numbers, int searchNum) {
                int count = 0;
                for (int i = 0; i < numbers.length; ++i) {
                        if (numbers[i] == searchNum)
                                count++;
                }
                return count;
        }

}
/********************************* End TaskImpl class ************************************/


/********************************* Start TaskRequestor  class ****************************/
package edu.unf.p2p.client;

import java.io.*;
import java.util.*;
import java.net.*;

import javax.xml.registry.*;
import javax.xml.registry.infomodel.*;

import edu.unf.p2p.common.Task;
import edu.unf.p2p.util.Log;
import edu.unf.p2p.util.RegistryUtil;

/**
 * This class is the main client program that spawns off threads to process the tasks
 */

public class TaskRequestor {
        private static String MY_NAME = "TaskRequestor";

        public static void main(String[] args) {
                if (args.length != 1 && args.length != 2) {
                        System.err.println("Usage: " + MY_NAME
                                        + " <total-data-len> <indiv-data-len>");
                        System.err.println("The total-data-len is the total length "
                                        + "of the data in KB to be processed by server threads "
                                        + "with each thread processing indiv-data-len KB data");
                        return;
                }
```

```
        String arg2 = args.length == 2 ? args[1] : null;
        //processMain(args[0], arg2);
        processMultiClients(args[0], arg2);
        //processPerfComp(args[0], arg2);
}

public static long processMain(String arg1, String arg2) {
        int dataLen = Integer.parseInt(arg1) * 1024;
        int indivDataLen = Integer.parseInt(arg2) * 1024;

        try {
                List serverList = getListFromSuperNodes();
                //indivDataLen = dataLen / serverList.size();
                //int numThreads = serverList.size();
                int numThreads = dataLen / indivDataLen;
                return process(indivDataLen, numThreads, serverList);
        } catch (IOException e) {
                System.err.println("Got IOException: " + e.getMessage());
        }
        return -1;
}

public static long processMultiClients(String arg1, String arg2) {
        int dataLen = Integer.parseInt(arg1) * 1024;
        int indivDataLen = Integer.parseInt(arg2) * 1024;

        try {
                for (int i = 0; i < 4; ++i) {
                        List serverList = getListFromSuperNodes();
                        int numThreads = dataLen / indivDataLen;
                        process(indivDataLen, numThreads, serverList);
                        System.out.println("-------Completed " + (i + 1)
                                        + " iteration ---------");
                }
        } catch (IOException e) {
                System.err.println("Got IOException: " + e.getMessage());
        }
        return -1;
}

public static long processPerfComp(String arg1, String arg2) {
        int dataLen = Integer.parseInt(arg1) * 1024;
        int indivDataLen = 128 * 1024;

        try {
                List serverList = getListFromSuperNodes();
                //indivDataLen = dataLen / serverList.size();
                int numThreads = dataLen / indivDataLen;
                return process(indivDataLen, numThreads, serverList);
        } catch (IOException e) {
                System.err.println("Got IOException: " + e.getMessage());
        }
        return -1;
}

public static long process(int indivDataLen, int numThreads, List serverList) {
        Log.logDebug(MY_NAME, "Data divided into " + numThreads
                        + " threads having data of size " + indivDataLen + " each");
        List notDoneThreads = new ArrayList();

        long startTime = System.currentTimeMillis();
        //long totalTime = 0;
        Map threadMap = new HashMap();
        Iterator i = null;
        for (int curThread = 0; curThread < numThreads;) {
                i = serverList.iterator();
                while (i.hasNext()) {
                        Map map = (Map) i.next();
                        String uri = (String) map.get("URI");
                        String serverThreads = (String) map.get("MAXTHREADS");
```

```
                        if (curThread == numThreads
                                        || Integer.parseInt(serverThreads) == 0)
                                continue;
                        Log.logDebug(MY_NAME, "Using map: " + map);
                        RequestHandler req = new RequestHandler(uri, indivDataLen,
                                        ++curThread);
                        req.start();
                        notDoneThreads.add(req);
                        map.put("MAXTHREADS", String.valueOf(Integer
                                        .parseInt(serverThreads) - 1));
                        threadMap.put(req, map);
                }
                i = notDoneThreads.iterator();
                while (i.hasNext()) {
                        RequestHandler req = (RequestHandler) i.next();
                        if (req.isDone()) {
                                i.remove();
                                //totalTime += req.getTimeTaken();
                                Map map = (Map) threadMap.get(req);
                                Log.logDebug(MY_NAME, "Freeing map: " + map);
                                String serverThreads = (String) map.get("MAXTHREADS");
                                map.put("MAXTHREADS", String.valueOf(Integer
                                                .parseInt(serverThreads) + 1));
                        }
                }
        }
        do {
                i = notDoneThreads.iterator();
                while (i.hasNext()) {
                        RequestHandler req = (RequestHandler) i.next();
                        if (req.isDone()) {
                                i.remove();
                                //totalTime += req.getTimeTaken();
                        }
                }
        } while (notDoneThreads.size() > 0);

        long endTime = System.currentTimeMillis();
        //System.out.println("TOTAL Processing Time to service the request: " +
        // totalTime + " msec");
        System.out.println("GRAND TOTAL Time to service the request: "
                        + (endTime - startTime) + " msec");
        return endTime - startTime;
}

public static URLConnection getConnection(String urlToConnect)
                throws Exception {
        URL url = new URL(urlToConnect);
        URLConnection con = url.openConnection();
        con.setDoInput(true);
        con.setDoOutput(true);
        con.setRequestProperty("Content-Type", "application/octet-stream;");
        return con;
}

public static List getListFromSuperNodes() throws IOException {
        List superNodes = RegistryUtil.getURLList("SuperNode");
        Log.logDebug(MY_NAME, "SuperNodes got back: " + superNodes);
        if (superNodes == null || superNodes.size() == 0)
                throw new IOException("No superNodes available");

        List serverList = new ArrayList();
        Iterator i = superNodes.iterator();
        while (i.hasNext()) {
                Map map = (Map) i.next();
                String superNodeURI = (String) map.get("URI");
                if (superNodeURI == null)
                        continue;
                try {
                        URLConnection con = getConnection(superNodeURI);
```

```
                                        BufferedInputStream in = new BufferedInputStream(con
                                                        .getInputStream());
                                        byte[] byteArr = new byte[1024];
                                        int numBytes = in.read(byteArr, 0, 1024);
                                        String str = new String(byteArr, 0, numBytes);
                                        Log.logDebug(MY_NAME, "Got str: " + str);
                                        if (str == null || "".equals(str))
                                                continue;

                                        StringTokenizer tokens = new StringTokenizer(str, "|");
                                        while (tokens.hasMoreTokens()) {
                                                String token = tokens.nextToken();
                                                StringTokenizer toks = new StringTokenizer(token, ";");
                                                Map newMap = new HashMap();
                                                newMap.put("URI", new String(toks.nextToken()));
                                                newMap.put("MAXTHREADS", new String(toks.nextToken()));
                                                serverList.add(newMap);
                                        }
                                } catch (Exception e) {
                                        e.printStackTrace();
                                        continue;
                                }
                        }

                        Log.logDebug(MY_NAME, "serverList: " + serverList);
                        return serverList;
                }
        }
/********************************** End TaskRequestor class **********************************/


/********************************** Start TaskHandler class **********************************/
package edu.unf.p2p.server;

import java.io.*;
import java.util.*;
import java.lang.reflect.*;

import javax.servlet.ServletException;
import javax.servlet.ServletConfig;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import edu.unf.p2p.util.Log;
import edu.unf.p2p.util.Constants;

/**
 * This class handles all requests passed to the server
 */

public class TaskHandler extends javax.servlet.http.HttpServlet {

        public static final String PROPERTIES_FILE = "TaskHandler";
        private static final int READ_BUF_SIZE = 1024;
        private static final int BUF_SIZE = 30 * 1024;
        private static final int FILE_WRITE_SUCCESS = 1;
        private static final int FILE_WRITE_FAIL_IO = 2;

        public void init(ServletConfig config) throws ServletException {
        }

        /**
         * Respond to a GET request to the servlet.
         * @param request The servlet request we are processing
         * @param response The servlet response we are producing
         *
         * @exception IOException if an input/output error occurs
         * @exception ServletException if a servlet error occurs
         */
```

```
public void doGet(HttpServletRequest request, HttpServletResponse response)
                throws IOException, ServletException {
        doPost(request, response);
}

public void doPost(HttpServletRequest request, HttpServletResponse response)
                throws IOException, ServletException {
        String cmd = request.getParameter("cmd");
        if (cmd == null) cmd = "status";

        if (cmd.equalsIgnoreCase("status")) {
                processStatus(request, response);
        } else if (cmd.equalsIgnoreCase("")) {
                processTask(request, response);
        }

}

public void processStatus(HttpServletRequest request,
                HttpServletResponse response) throws IOException, ServletException {
        OutputStream out = response.getOutputStream();
        out.write(Constants.STATUS_ACTIVE);
        out.close();
        //Log.logDebug(this, "Sent status: " + Constants.STATUS_ACTIVE);
        return;
}

public void processTask(HttpServletRequest request,
                HttpServletResponse response) throws IOException, ServletException {
        byte[] bufArr = new byte[BUF_SIZE];
        Log.logDebug(this, "Processing task");
        try {
                InputStream in = request.getInputStream();

                ObjectInputStream numIn = new ObjectInputStream(in);
                int numBytes = numIn.readInt();
                int actualBytes = in.read(bufArr, 0, numBytes);
                if (actualBytes != numBytes) {
                        throw new IOException("Error reading class, got " + actualBytes
                                        + " instead of " + numBytes);
                }
                Log.logDebug(this, "Actual bytes: " + actualBytes);

                MyClassLoader loader = new MyClassLoader();
                loader.b = bufArr;
                loader.offset = 0;
                loader.len = numBytes;

                loader.getObj(in, response.getOutputStream());
        } catch (Exception e) {
                e.printStackTrace();
                System.out.println("got exception (any): " + e.getMessage());
                return;
        }

}

public class MyClassLoader extends ClassLoader {
        public int offset, len;

        public byte[] b;

        private Hashtable classes = new Hashtable();

        public Class loadClass(String name) throws ClassNotFoundException {
                System.out.println("came inside loadclass");
                return (loadClass(name, true));
        }

        public Object getObj(InputStream in, OutputStream out)
```

```
                throws IOException, ClassNotFoundException, Exception {
        Class c = loadClass("edu.unf.p2p.server.ObjDeserializer", true);
        try {
                Class[] parameterTypes = new Class[] { InputStream.class,
                                OutputStream.class };
                Object[] arguments = new Object[] { in, out };

                parameterTypes = new Class[] {};
                arguments = new Object[] {};
                Constructor con = c.getConstructor(parameterTypes);
                Object obj_in = con.newInstance(arguments);

                parameterTypes = new Class[] { InputStream.class,
                                OutputStream.class };
                arguments = new Object[] { in, out };
                Method readMethod = c.getMethod("deserialize", parameterTypes);
                Log.logDebug(this, "Invoking method");
                readMethod.invoke(obj_in, arguments);
                Log.logDebug(this, "Done");
                return null;
        } catch (NoSuchMethodException e) {
                System.out.println(e + ": " + e.getMessage());
        } catch (IllegalAccessException e) {
                System.out.println(e + ": " + e.getMessage());
        } catch (InvocationTargetException e) {
                String mesg = e.getTargetException().getMessage();
                System.out.println(e + ": " + mesg);
                e.printStackTrace();
        }

        /*
         * Object obj_in = new java.io.ObjectInputStream(in); Object task =
         * obj_in.readObject();
         */
        return null;
}

/**
 * This is the required version of loadClass which is called both from
 * loadClass above and from the internal function FindClassFromClass.
 */
public synchronized Class loadClass(String className, boolean resolveIt)
                throws ClassNotFoundException {
        Class result;
        byte classData[];

        ClassLoader defaultLoader = this.getClass().getClassLoader();
        System.out.println("      >>>>>> Load class : " + className);

        /* Check our local cache of classes */
        result = (Class) classes.get(className);
        if (result != null) {
                System.out.println("      >>>>>> returning cached result: "
                                + className);
                return result;
        }

        /* Check with the primordial class loader */
        try {
                result = super.findSystemClass(className);
                System.out
                                .println("      >>>>>> returning system class (in CLASSPATH): "
                                                + className);
                return result;
        } catch (ClassNotFoundException e) {
                System.out.println("      >>>>>> Not a system class: "
                                + className);
        }

        try {
```

```
                        if (className.equals("edu.unf.p2p.server.ObjDeserializer")) {
                                result = Class.forName(className);
                                byte[] byteArr = loadFileBytes(className);
                                if (byteArr == null)
                                        throw new ClassNotFoundException("class not found");
                                result = defineClass(byteArr, 0, byteArr.length);
                                System.out
                                                .println("      >>>>>> returning --- class (from
class.forname): "
                                                                + className);
                                return result;
                        }
                        result = defaultLoader.loadClass(className);
                        System.out
                                        .println("      >>>>>> returning other class (in CLASSPATH): "
                                                        + className);
                        return result;
                } catch (ClassNotFoundException e) {
                        System.out.println("      >>>>>> Not a system class: "
                                        + className);
                }

                System.out.println("      >>>>>> didn't find " + className);

                /* Try to load it from our repository */
                classData = b;
                if (classData == null) {
                        throw new ClassNotFoundException();
                }

                System.out.println("      >>>>>> didn't find " + className
                                + " in repository.");

                /* Define it (parse the class file) */
                result = defineClass(classData, offset, len);
                if (result == null) {
                        throw new ClassFormatError();
                }

                if (resolveIt) {
                        resolveClass(result);
                }

                classes.put(className, result);
                System.out.println("      >>>>>> Returning newly loaded class: "
                                + className);
                return result;
        }

        /**
         * Search the zip file bytes, and return an array of bytes corresponding
         * to the given class name
         */
        private byte[] loadFileBytes(String className) {
                try {
                        Class taskClass = this.getClass();
                        ClassLoader loader = taskClass.getClassLoader();
                        InputStream inStream = loader

.getResourceAsStream("edu/unf/p2p/server/ObjDeserializer.class");
                        BufferedInputStream in = new BufferedInputStream(inStream);

                        byte[] byteArr = new byte[BUF_SIZE];
                        int numBytes = in.read(byteArr, 0, BUF_SIZE);
                        if (numBytes == -1) {
                                Log.logDebug(this, "ERROR: filesize > " + BUF_SIZE);
                                return null;
                        }
                        byte[] classBytes = new byte[numBytes];
                        for (int i = 0; i < numBytes; i++)
```

```
                                        classBytes[i] = byteArr[i];
                                return classBytes;
                        } catch (IOException e) {
                                e.printStackTrace();
                                return null;
                        }
                }
        }

}
/********************************* End  TaskHandler class **********************************/


/********************************* Start  ObjDeserializer class ********************************/
package edu.unf.p2p.server;

import java.io.*;
import java.util.*;
import java.lang.reflect.*;

import edu.unf.p2p.common.Task;
import edu.unf.p2p.util.Log;
import edu.unf.p2p.util.Constants;

/**
 * This class deserializes the object from client
 */

public class ObjDeserializer {
        private static final String MY_NAME = "ObjDeserialier";
        private static final int READ_BUF_SIZE = 1024;
        private static final int BUF_SIZE = 30 * 1024;
        private static final int FILE_WRITE_SUCCESS = 1;
        private static final int FILE_WRITE_FAIL_IO = 2;

        public ObjDeserializer() {
                Log.logDebug(MY_NAME, "inside constructor");
        }

        public static void deserialize(InputStream in, OutputStream out)
                        throws IOException {
                byte[] bufArr = new byte[BUF_SIZE];
                int numBytes = -1;
                try {
                        ObjectInputStream objIn = new ObjectInputStream(in);
                        Task task = (Task) objIn.readObject();

                        task.process();

                        Log.logDebug(MY_NAME, "Read from client");

                        ObjectOutputStream objOut = new ObjectOutputStream(out);
                        objOut.writeObject(task);

                        objOut.close();
                        in.close();
                        out.close();
                } catch (Exception e) {
                        e.printStackTrace();
                        System.out.println("got exception (any): " + e.getMessage());
                        return;
                }

                Log.logDebug(MY_NAME, "Finished servicing event ");
        }
}
/********************************* End ObjDeserializer class **********************************/
```

```
/******************************** Start ServerMonitor class ********************************/
package edu.unf.p2p.supernode;

import java.text.SimpleDateFormat;
import java.io.*;
import java.net.URL;
import java.net.URLConnection;
import java.util.ArrayList;
import java.util.Calendar;
import java.util.Date;
import java.util.HashMap;
import java.util.Iterator;
import java.util.List;
import java.util.Map;
import java.util.Set;

import edu.unf.p2p.util.Log;
import edu.unf.p2p.util.Constants;
import edu.unf.p2p.util.RegistryUtil;

/**
 * Class to monitor server on the supernode side
 *
 * @author Jayant Mishra
 *
 */
public class ServerMonitor extends Thread {
        private static final String MY_NAME = "ServerMonitor";
        private static long monitorInterval = 10L; // in minutes
        private static List allURLs;
        private static List activeURLs;

        public static void main(String args[]) {
                refreshActiveList();
        }

        public ServerMonitor() {
                Log.logDebug(this, "In constructor of ServerMonitor");
        }

        public void run() {
                while (true) {
                        refreshActiveList();
                }
        }

        public static void refreshActiveList() {
                try {
                        // get list of URLs
                        List newActiveURLs = new ArrayList();
                        //Log.logDebug(MY_NAME, "Calling getURLList method of
                        // RegistryUtil");
                        allURLs = RegistryUtil.getURLList("Power server service");
                        Log.logDebug(MY_NAME, "Got back: " + allURLs);
                        if (allURLs == null)
                                throw new IOException("No server URLs");

                        // get status from each URL
                        Iterator i = allURLs.iterator();
                        while (i.hasNext()) {
                                Map map = (Map) i.next();
                                String URL = (String) map.get("URI");
                                String numThreads = (String) map.get("MAXTHREADS");
                                Log.logDebug(MY_NAME, "Getting status from URL: " + URL);
                                URL url = new URL(URL + "?cmd=status");
                                URLConnection con = url.openConnection();
                                con.setUseCaches(false);
                                con.setDoInput(true);
                                con.setDoOutput(true);
                                con.setRequestProperty("Content-Type",
```

```
                                        "application/octet-stream;");
                        con.connect();

                        OutputStream out = con.getOutputStream();
                        InputStream in = con.getInputStream();
                        int retCode = in.read();
                        Log.logDebug(MY_NAME, "Got back status code: " + retCode
                                        + " from URL: " + URL);
                        if (retCode == Constants.STATUS_ACTIVE) {
                                // add to list of active URLs
                                newActiveURLs.add(URL + ";" + numThreads);
                        }
                }
                activeURLs = newActiveURLs;

        } catch (IOException e) {
                Log.logDebug(MY_NAME, "No server URLs found");
        } finally {
                try {
                        Thread.sleep(monitorInterval * 60 * 1000);
                } catch (InterruptedException e) {
                        Log.logWarn(MY_NAME, "InterruptedException encountered; "
                                        + e.getMessage());
                }
        }
}

/**
 * Returns the monitorInterval.
 *
 * @return long
 */
public static long getMonitorInterval() {
        return monitorInterval;
}

public static List getActiveURLs() {
        return activeURLs;
}

/**
 * @param l
 */
public static void setMonitorInterval(long l) {
        monitorInterval = l;
}

}
/********************************** End ServerMonitor class **********************************/


/********************************** Start SuperNode class **********************************/
package edu.unf.p2p.supernode;

import java.io.*;
import java.util.*;
import java.lang.reflect.*;

import javax.servlet.ServletException;
import javax.servlet.ServletConfig;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import edu.unf.p2p.util.Log;
import edu.unf.p2p.util.Constants;

/**
 * This class handles all requests passed to the supernode
 */
```

```
public class SuperNode extends javax.servlet.http.HttpServlet {

	private static final int READ_BUF_SIZE = 1024;
	private static final int BUF_SIZE = 30 * 1024;
	private static final int FILE_WRITE_SUCCESS = 1;
	private static final int FILE_WRITE_FAIL_IO = 2;

	public void init(ServletConfig config) throws ServletException {
		ServerMonitor monitor = new ServerMonitor();
		monitor.start();
		Log.logDebug(this, "Started monitor");
	}

	/**
	 * Respond to a GET request to this servlet.
	 *
	 * @param request The servlet request we are processing
	 * @param response The servlet response we are producing
	 *
	 * @exception IOException if an input/output error occurs
	 * @exception ServletException if a servlet error occurs
	 */
	public void doGet(HttpServletRequest request, HttpServletResponse response)
			throws IOException, ServletException {
		doPost(request, response);
	}

	public void doPost(HttpServletRequest request, HttpServletResponse response)
			throws IOException, ServletException {
		String cmd = request.getParameter("cmd");
		if (cmd == null)
			cmd = "getActiveURLs";

		if (cmd.equalsIgnoreCase("getActiveURLs")) {
			processGetActiveURLs(request, response);
		}
		Log.logDebug(this, "Finished servicing event: " + cmd);
	}

	public void processGetActiveURLs(HttpServletRequest request,
			HttpServletResponse response) throws IOException, ServletException {
		List URLs = ServerMonitor.getActiveURLs();
		StringBuffer URLStrBuff = new StringBuffer();
		Iterator i = URLs.iterator();
		while (i.hasNext()) {
			String URL = (String) i.next();
			URLStrBuff.append(URL + "|");
		}
		OutputStream out = response.getOutputStream();
		out.write((URLStrBuff.toString()).getBytes());
		out.close();
		return;
	}

}
/********************************** End SuperNode class ***********************************/


/********************************** Start Constants class ***********************************/
package edu.unf.p2p.util;

/**
 * @author Jayant Mishra
 */
public class Constants {

	public static final int BUF_SIZE = 8 * 1024;

	public static final int STATUS_DEAD = -1;
	public static final int STATUS_INACTIVE = 0;
```

```
	public static final int STATUS_ACTIVE = 1;
	public static final int STATUS_BUSY = 2;

}
/********************************* End Constants class ************************************/


/********************************* Start Log class ************************************/
package edu.unf.p2p.util;

import java.util.*;
import java.io.*;
import java.text.SimpleDateFormat;
import javax.xml.parsers.*;
import org.w3c.dom.*;

/**
 * @author Jayant Mishra
 */
public class Log {
	public static final String LOG4J_CAT_FATAL = "FATAL";
	public static final String LOG4J_CAT_ERROR = "ERROR";
	public static final String LOG4J_CAT_WARN = "WARN";
	public static final String LOG4J_CAT_INFO = "INFO";
	public static final String LOG4J_CAT_DEBUG = "DEBUG";
	public static final String LOG4J_CATEGORY_STR = "log4j.rootCategory";

	private static final String MY_NAME = "Log";

	private static final List LOG4J_CAT_ARRAY = new ArrayList();
	static {
		LOG4J_CAT_ARRAY.add(LOG4J_CAT_DEBUG);
		LOG4J_CAT_ARRAY.add(LOG4J_CAT_INFO);
		LOG4J_CAT_ARRAY.add(LOG4J_CAT_WARN);
		LOG4J_CAT_ARRAY.add(LOG4J_CAT_ERROR);
		LOG4J_CAT_ARRAY.add(LOG4J_CAT_FATAL);
	};

	private static int curLogLevel = LOG4J_CAT_ARRAY.indexOf(LOG4J_CAT_DEBUG);

	public static void log(String name, String mesg, int logLevel) {
		if (curLogLevel <= logLevel) {
			SimpleDateFormat dateFormat = new SimpleDateFormat(
					"yyyy-MM-dd HH:mm:ss");
			Date date = new Date();
			System.out.println(dateFormat.format(date) + " : " + name + " - "
					+ mesg);
		}
	}

	public static void logFatal(Object obj, String mesg) {
		log((obj instanceof String ? (String) obj : obj.getClass().getName()),
				mesg, LOG4J_CAT_ARRAY.indexOf(LOG4J_CAT_FATAL));
	}

	public static void logError(Object obj, String mesg) {
		log((obj instanceof String ? (String) obj : obj.getClass().getName()),
				mesg, LOG4J_CAT_ARRAY.indexOf(LOG4J_CAT_ERROR));
	}

	public static void logWarn(Object obj, String mesg) {
		log((obj instanceof String ? (String) obj : obj.getClass().getName()),
				mesg, LOG4J_CAT_ARRAY.indexOf(LOG4J_CAT_WARN));
	}

	public static void logInfo(Object obj, String mesg) {
		log((obj instanceof String ? (String) obj : obj.getClass().getName()),
				mesg, LOG4J_CAT_ARRAY.indexOf(LOG4J_CAT_INFO));
	}
```

```
        public static void logDebug(Object obj, String mesg) {
                log((obj instanceof String ? (String) obj : obj.getClass().getName()),
                                mesg, LOG4J_CAT_ARRAY.indexOf(LOG4J_CAT_DEBUG));
        }

}
/******************************** End Log class ****************************************/


/******************************** Start RegistryUtil class ****************************/
package edu.unf.p2p.util;

import javax.xml.registry.*;
import javax.xml.registry.infomodel.*;

import java.io.*;
import java.util.*;

public class RegistryUtil {
        private static final String QUERY_URL = "query.url";
        private static final String PUBLISH_URL = "publish.url";
        private static final String PROXY_HOST = "http.proxy.host";
        private static final String PROXY_PORT = "http.proxy.port";
        public static final String PROPERTIES_FILE = "TaskHandler";

        public static void main(String args[]) throws Exception {
                List serverURL = RegistryUtil.getURLList("Power server service");
                Log.logDebug("", "Got back: " + serverURL);
        }

        public static List getURLList(String svcName) throws IOException {
                Properties props = new Properties();
                ResourceBundle bundle = ResourceBundle.getBundle(PROPERTIES_FILE);
                //String svcName = "Power server service";
                List list = null;
                try {
                        //props.load(new FileInputStream(PROPERTIES_FILE));
                        list = RegistryUtil.executeQueryTest(bundle, svcName);
                        if (list == null)
                                throw new IOException("Got NO companies offering " + svcName);
                } catch (JAXRException e) {
                        System.err.println("Error during the test: " + e.getMessage());
                        throw new IOException(e.getMessage());
                } catch (IOException e) {
                        System.err.println("Can not open properties file: "
                                        + e.getMessage());
                        throw e;
                }
                return list;
        }

        public static List executeQueryTest(ResourceBundle bundle, String svcName)
                        throws JAXRException {
                List retList = new ArrayList();
                try {
                        Properties connProps = setConnectionProperties(bundle);

                        ConnectionFactory factory = ConnectionFactory.newInstance();
                        factory.setProperties(connProps);
                        Connection conn = factory.createConnection();
                        RegistryService rs = conn.getRegistryService();
                        BusinessQueryManager bqm = rs.getBusinessQueryManager();
                        BusinessLifeCycleManager blcm = rs.getBusinessLifeCycleManager();

                        ClassificationScheme cScheme = bqm.findClassificationSchemeByName(
                                        null, "ntis-gov:naics");
                        Classification classification = blcm.createClassification(cScheme,
                                        "Other Computer Related Services", "541519");
                        Collection classifications = new ArrayList();
                        classifications.add(classification);
```

```
                    // make JAXR request
                    BulkResponse response = bqm.findOrganizations(null, null,
                                    classifications, null, null, null);
                    Collection orgs = response.getCollection();

                    Iterator orgIter = orgs.iterator();
                    while (orgIter.hasNext()) {
                            Organization org = (Organization) orgIter.next();
                            /*
                             * System.out.println("Organization Name: " + getName(org));
                             * System.out.println("Organization Key: " +
                             * org.getKey().getId()); System.out.println("Organization
                             * Description: " + getDescription(org));
                             */
                            Collection services = org.getServices();
                            Iterator svcIter = services.iterator();
                            while (svcIter.hasNext()) {
                                    Service service = (Service) svcIter.next();
                                    String name = getName(service);
                                    if (name != null && !name.equals(svcName))
                                            continue;
                                    String desc = getDescription(service);
                                    /*
                                     * System.out.println("\tService Name: " +
                                     * getName(service)); System.out.println("\tService Key: " +
                                     * service.getKey().getId()); System.out.println("\tService
                                     * Description: " + getDescription(service));
                                     */

                                    // Get a collection of ServiceBindings from a Service
                                    Collection serviceBindings = service.getServiceBindings();
                                    // Iterate through the collection to get an individual
                                    // ServiceBinding
                                    Iterator sbIter = serviceBindings.iterator();
                                    String uri = "";
                                    while (sbIter.hasNext()) {
                                            ServiceBinding serviceBinding = (ServiceBinding) sbIter
                                                            .next();
                                            // Get URI of the service. You can access the service
                                            // through this URI.
                                            uri = serviceBinding.getAccessURI();
                                    }
                                    Map map = new HashMap();
                                    map.put("URI", new String(uri));
                                    map.put("MAXTHREADS", new String(desc));

                                    retList.add(map);
                            }
                    }
            } catch (JAXRException e) {
                    e.printStackTrace();
            }
            return retList;

}

private static Properties setConnectionProperties(ResourceBundle bundle) {
            String httpProxyHost = "";
            String httpProxyPort = "";
            String regUrli = "";
            String regUrlp = "";

            String temp;

            //temp = ((String)props.getProperty(QUERY_URL)).trim();
            temp = (bundle.getString(QUERY_URL)).trim();
            if (temp != null)
                    regUrli = temp;

            //temp = ((String)props.getProperty(PUBLISH_URL)).trim();
```

```
                temp = (bundle.getString(PUBLISH_URL)).trim();
                if (temp != null)
                        regUrlp = temp;

                //temp = ((String)props.getProperty(PROXY_HOST)).trim();
                temp = (bundle.getString(PROXY_HOST)).trim();
                if (temp != null)
                        httpProxyHost = temp;

                //temp = ((String)props.getProperty(PROXY_PORT)).trim();
                temp = (bundle.getString(PROXY_PORT)).trim();
                if (temp != null)
                        httpProxyPort = temp;

                Properties connProps = new Properties();
                connProps.setProperty("javax.xml.registry.queryManagerURL", regUrli);
                connProps
                                .setProperty("javax.xml.registry.lifeCycleManagerURL", regUrlp);
                connProps.setProperty("javax.xml.registry.factoryClass",
                                "com.sun.xml.registry.uddi.ConnectionFactoryImpl");
                connProps.setProperty("com.sun.xml.registry.http.proxyHost",
                                httpProxyHost);
                connProps.setProperty("com.sun.xml.registry.http.proxyPort",
                                httpProxyPort);
                return connProps;
        }

        private static String getName(RegistryObject ro) throws JAXRException {
                try {
                        return ro.getName().getValue();
                } catch (NullPointerException npe) {
                        return "";
                }
        }

        private static String getDescription(RegistryObject ro)
                        throws JAXRException {
                try {
                        return ro.getDescription().getValue();
                } catch (NullPointerException npe) {
                        return "";
                }
        }
}
/******************************** End RegistryUtil class **********************************/
```

VITA

Jayant Mishra has a Bachelor of Science degree from the Regional Engineering College, Bhopal, India, 1996, and expects to receive a Master of Science in Computer and Information Sciences from the University of North Florida in 2006. Dr. Sanjay Ahuja of the University of North Florida served as Jayant's thesis adviser. Jayant is a Senior IT Consultant, working for CSX, Inc. for the past three years. Prior to that, Jayant worked as an IT Consultant for Citicards, N.A. for two years.

Jayant has on-going interests in distributed computing, peer-to-peer networks, and object-oriented software design. Jayant has extensive programming experience in C, C++ and Java languages. Additionally, Jayant has strong knowledge of Java 2 Enterprise Edition (J2EE), Model View Controller (MVC), Design Patterns, and Web Services application development. Jayant is fluent in Hindi and enjoys tennis. Married for the past eight years, Jayant has one son, age seven years and one daughter, age one year.