Peer-to-Peer Distributed Computing Applied to Prototypical Problems.
by

Shirish Ranjit*

April 16, 2004

A Thesis submitted to the Faculty
of the Gradute School,
Marquette Univeristy
in Partial Fulfillment of
the Requirements for
the Degree of
Master of Science in Computing.

Milwaukee, Wisconsin
May, 2004

*Marquette University, P.O. Box 1881, Milwaukee, WI 53201-1881

Abstract

The goal of the project is to develop a Peer-to-Peer (P2P) Distributed Computing environment, named Octopus, such that we can capitalize unused computer resources to solve computing intensive problems as well as simple problems. The main features of the architecture are

1. Solve large scale problems
2. Capitalized unused computer resources (wired and wireless)
3. Be independent of problem type
4. Scalable
5. Be fault tolerant and available

The Octopus is designed to solve a problem by pulling idle resources from the network of computers that are either wired or wireless (mobile). The Octopus architecture is independent of the type of problem. It can solve a large-scale problem that requires a lot of computing resources as well as small requests. For example, when users are not using the desktops in an office, the processing time of those computers can be used to solve a problem that benefits the company.

A processing intensive problem is a prototypical problem for this type of computing. For example, a portfolio optimization problem is computing intensive. With the Octopus, we can optimize buys and sells from a universe of stocks that maximizes the return of the portfolio. We solve the problem with idle resources in the enterprise rather than with a dedicated large expensive server.

In essence, in a P2P Distributed Computing environment, the data for the problem is sliced into small units of work (work unit) and sent to different computers (volunteers). These volunteers do not know the requestors of work unit or their peers. A publisher, a volunteer, publishes the work unit into a space. Volunteers with idle resources collect a work unit from the space to solve the problem. These volunteers process according to the instructions they receive in the work unit. Once their unit of work is complete, the results are published into the space. The publisher acquires solutions from the space. This environment is like a group of computers working to solve a problem with little knowledge about their peers.

We take couple of problems to demonstrate that the prototype of the Octopus architecture. The problems that we chose are finding a stock quote and a Monte Carlo simulation of value of a simple stock portfolio. We choose two different types of problem to demonstrate that the Octopus architecture is truly independent of the type of problems.

We have designed, developed, and demonstrated that it is a formidable solution for common business problems by using wired or wireless resources. Our simple demonstration has shown that the Octopus architecture is a viable framework for P2P Distributed Computing. Though the current demonstration version contains a minimal set of function, we made a case that this is a viable product for solving business problem and managing corporate information.

Acknowledgements This paper is dedicated to my family who has provided constant support during this endeavor. Specially my wife and brother, who have endured my never ending phrase "almost complete" for more than two years. I am in debt for their support and encouragements. My family has been the source of my motivation.

I want to thank the committee for their time, support, and their motivation. This work would not have been possible without the level of support that I received from my advisor, Dr. George Corliss. There is no word that can express my gratitude for his constant motivation, academic advising, and the support that he gave me with endless revision of this paper. I am in debt for teaching me not only mathematics and computing but also English writing. I would like to acknowledge the level of commitment that Dr. Corliss provided me in finishing this paper.

In last four years, many friends have shaped my work. I would like to thank all of them for their contributions. I especially thank Melaine Anger for her critical review and superb suggestions. I appreciate everybody's support that made this work possible.

A patent is pending for the Octopus architecture, framework, and implementation.

Contents

1 Introduction

Peer-to-Peer (P2P) networks and Distributed Computing are topics that interest the scientific and Internet communities. Since the birth of the Internet, Peer-to-Peer (P2P) networking and Distributed Computing have been topics for research. Similarly, Financial Portfolio Optimization has been a topic of research in the financial communities.

In this paper, we develop P2P distributed computing to solve a problem. Here, we take a stock quote problem and a Monte Carlo simulation of a financial portfolio to demonstrate an implementation of our P2P distributed computing system.

In this chapter, we give a top-down overview of the layers of services that comprise our system. We can organize our system in four primary layers of services:

- Problem: any CPU-consuming problem. For the purpose of illustration, we take a interesting Financial Problem (FP).
- Peer-to-Peer distributed computing system, 'Octopus:' this layer consists of two sub-layers as shown in Figure 1.
  - Distributed Communication Client (DCC).
  - Distributed Communication Space (DCS).
- JavaSpace
- Jini and RMI

In subsequent sections of this chapter, we give a high-level overview of each layer. Then, we discuss each layer in detail in subsequent chapters.

In this introductory chapter, we have given a top-down overview of the layers that comprise a P2P Distributed Computing environment. In the following chapters, we take a bottom up approach to describe the proposed architecture, building layers from the base to upper layer. In Chapter 2, we give an overview of relevant research publications in P2P networking, distributed Computing and financial optimization. Then, we start assembling the building blocks of our P2P Distributed Computing environment, and

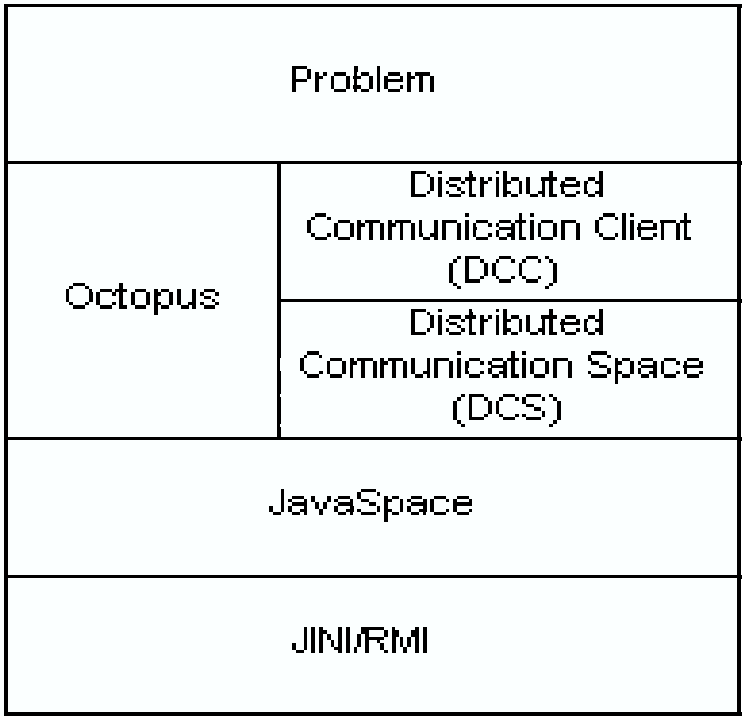

Figure 1: Layers and Services

construct the environment. We start with the technology base, Jini and RMI, in Chapter 3. Then, we describe JavaSpace technology in Chapter 4. Once the technology foundation has been laid, we present Octopus and its components DCS and DCC in Chapter 5. At this point, we now have an environment for P2P Distributed Computing. The final layer is that of the problem. A FP is developed and described in Chapter 6. In Chapter 7, we present the system benefits and risks. We give a detailed step-by-step description of the Octopus implementation in Chapter 8. We present our conclusions and extensions of the Octopus model and possible future directions in the Chapter 9.

1.1 Problem Layer

Financial 'Engineers' employ various mathematical tools for analyzing large amounts of data. They explore not only mathematical and statistical models but also Artificial Neural Network (ANN) models. These models are used for valuations and forecasting of financial instruments.

We choose a problem to illustrate our P2P and distributed computing architecture and to demonstrate that the Octopus framework works. We chose a stock quote problem and a Monte Carlo simulation of a simple stock portfolio. From here on, we refer to these as Financial Problems (FP). We describe the problem in detail in Chapter 6.

1.2 Octopus Layer

In this section, we give definitions of P2P networking, distributed computing, and P2P distributed computing. Then, we give a brief overview of Octopus, our architecture for P2P distributed computing. Before going into details of the Octopus system in Chapter 5, we first give a brief background of Peer-to-Peer (P2P) networking and Distributed Computing.

1.2.1 Peer-to-Peer Networking

Peer-to-Peer (P2P) networking is an architecture that supports sharing of resources and services among a collection of computers. Although most of the Computer Science communities might agree with the previous statement, there is not an industry standard definition for P2P networking. Schollmeier [20] gave definitions for P2P networking. He gave a general definition from which he classified P2P networking further into two sub-definitions:

Definition 1 *[20] Peer-to-Peer (P2P) networking is a community of computers that shares resources such as processing power, storage, contents, and so on. Shared resources are directly available to all participants in the community without passing through intermediary entities. That is, there is no server as in client-server models. Here every participant is a publisher of services and also a consumer of services.*

P2P networking can be further classified into two sub-definitions [20]. They are *"Pure"* and *"Hybrid"* P2P networking:

Definition 2 *[20] A network is a "pure" P2P network if it satisfies definition 1, and if any single arbitrarily chosen entity in the network can be removed without hampering the network's ability to provide all services.*

Definition 3 *[20] A network is a "hybrid" P2P network if it satisfies definition 1, and a central entity is necessary to provide parts of the offered network services.*

We discuss Schollmeier's definitions further in Chapter 2. In the following section, we define Distributed Computing. Then in the subsequent section, we give a definition of P2P distributed computing.

1.2.2 Distributed Computing

'Distributed Computing' is a collective way of working on a problem with a network of computers. It involves sending pieces of a problem to a number of computers in a network to achieve a 'single goal.' A 'single goal' means either finding a result of a problem or executing an algorithm. In the scientific community, distributed computing involves decomposing a problem into smaller pieces such that a small, less powerful computer can process the information to achieve the single goal. Most distributed computing models involve at least a server that manages data, connections to computers, and a community of computers that are analyzing the data. For example, a search on the Internet can take a long time for a single computer. Similarly, finding a signal transmission from an alien source in a continuous data stream from a radio telescope can take many years for a single super-computer. However, by decomposing a problem and distributing the work, many computers can be use to solve the problem.

An example of a well known distributed system is The Search for Extraterrestrial Intelligence (SETI) project. SETI@HOME is a distributed computing application, which allows computers on the Internet to join the SETI project in searching for evidence of signals from outer space. The SETI distributed computing system decomposes the problem of searching for identifiable signals in radio-telescope data into smaller pieces

and distributes them to different computers on the Internet for analysis. After the completion of their work, these computers send back their results to SETI for further analysis. The SETI project is discussed further in the Chapter 2.

We have given a definition of P2P networking and distributed computing. We define P2P Distributed Computing in the following subsection.

1.2.3 Peer-to-Peer Distributed Computing

Peer-to-Peer (P2P) Distributed Computing uses a P2P networking model to do distributed computing. In other words, using the P2P model, we distribute pieces of a problem and related data to a community of computers in such a way that those computers do not have to know who is requesting services. The P2P Distributed Computing model has 'Publishers' and 'Acquirers.' A publisher's role is to provide requests, for example a portion of a FP problem, while an acquirer's role is to provide services to requests, for example a solution to portion of a FP problem. We define requests as 'Work Units.'

A 'Work Unit' is a portion of a whole problem with the property that it can be processed independently. In a P2P Distributed Computing environment, it is the responsibility of a publisher to provide a work unit and the responsibility of an acquirer to service the request.

A P2P Distributed Computing model does not have a 'server' and a 'servant' concept as in Distributed Computing. The P2P Distributed Computing model has publishers and acquirers. A publisher is the one who is requesting services from other peers in the community. Similarly, acquirers are those peers who are providing services. In this model, anyone can be a publisher, and anyone can be an acquirer.

We call our implementation of the P2P Distributed Computing model **'Octopus.'** Octopus has two sub-layers. They are the **'Distributed Communication Client (DCC)'** and the **'Distributed Communication Space (DCS)'** layers. The Distributed Communication Client layer is an application that resides in peer computers.

DCC provides services to communicate among peers. The Distributed Communication Space (DCS) layer is a communication hub, residing in a peer computer, providing services to all peers for publishing work units and consuming work units. Hence, peers communicate work units and results through DCS.

The system is analogous to the structure of an octopus, in the sense that it has a central body with many tentacles. DCC applications (peer computers) are like the tentacles of the octopus, and DCS (a JavaSpace) is the central body. Figure 2 illustrates the P2P Distributed Computing architecture.

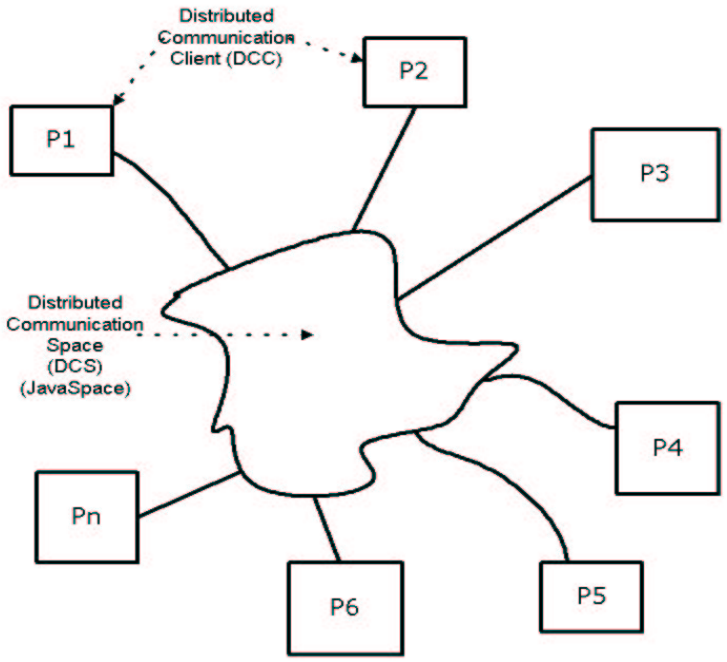

Figure 2: Octopus, the Peer-to-Peer Distributed Computing Architecture

Octopus is our P2P distributed computing system, which is a vehicle for distributing problems defined in a problem layer among volunteer computers. It is also a medium through which volunteer computers communicate solutions to publishers of problems. Architecting and developing Octopus is the focus of this thesis.

Octopus' components DCS and DCC depend on a JavaSpace layer and a Jini/RMI layer. DCS uses a JavaSpace layer as a communication hub. It is in this JavaSpace layer

that publishers request services and acquirers search for these requests. A JavaSpace is a Jini service.

In the following sections, we provide a brief introduction to JavaSpace, Jini, and RMI.

1.3 JavaSpace

In this section, we give a brief introduction to Sun Microsystems' JavaSpace technology; we discuss JavaSpace in detail in Chatper 4, along with additional key specifications.

A JavaSpace is a network-accessible "shared-memory" [7] space in which remote processors can access (write or take) objects concurrently. Octopus' Distributed Communication Space (DCS) is an example of an implementation of a JavaSpace. All the peers publish their work units and results in this JavaSpace. They also search for work units and results for consumption here.

Sun Microsystems provides not only the specification for JavaSpace, but also its implementation. We have used this implementation in Octopus. JavaSpace is built as Jini services. Communication is provided through RMI. We introduce Jini and RMI in the following section.

1.4 Jini/RMI

In this section, we introduce Sun Microsystems' Jini and RMI technologies. Jini and RMI are the foundations of JavaSpace services. Jini technology defines a set of application programming interfaces (APIs) and network protocols to build and deploy distributed systems. RMI is a means of communication between these Java applications.

Jini specifies a set of APIs and network protocols for building distributed systems that are organized as federations of services. A federation of services is a set of available services on the network that a consumer can request to accomplish a task [22]. The concept of 'Service' is the most important in Jini technology because a service is

a basic entity that can be consumed or published by hardware devices, software, communications channels, human users, or other services. For example, a Jini enabled disk drive could offer a 'storage' service. We discuss Jini technology in detail in Chapter 3.

RMI (Remote Method Invocation) specifies a set of APIs for applications to invoke processing on remote objects and to facilitate sharing of resources across systems throughout a network. The ability to invoke a 'method' on a remote object offers a model for distributing Java objects in a network. A 'method' is a function in an object designed to accomplish a pre-specified task. Therefore, RMI specifications provide a framework for communication among applications to accomplish a single goal. We discuss relevant specifications of RMI in Chapter 3.

We discussed the overview of the paper in this chapter. We present current literature reviews in the following chapter.

2 Literature Review

In this chapter, we survey the literature of Peer-to-Peer networking, distributed computing, Java technology (J2EE, Jini, JavaSpace), and financial portfolio theory. The intent of this chapter is to give an overview of research and development in these respective fields.

2.1 Peer-to-Peer (P2P) Networking

Peer-to-Peer Networking has been a topic of research in the networking community focusing on networking for collaborative use of computing resources. The collaborations include file sharing, storage resource sharing, and CPU resource sharing. In this section, we survey some of relevant research.

The Communications of the ACM devoted the first half of the February 2003 issue on the P2P Networking articles. This is an indication of the importance of the topic in the industry. The special issue had viewpoints, survey articles, and a few technical articles. We discuss few of those articles below.

The viewpoint by Schoder and Fischbach [19] presented P2P networking and also raised issues of network control, costs, security, interoperability, and metadata. In the paper, we raise those issues and discuss mitigating strategies.

Agre [1] discussed socio-economic and political impacts due to the use of P2P networking. Agre presented four theories of relationship between political institutions, socio-economic structures, and information technology. He especially cited Thorstein Veblen, a Norweigian-American economist, and Friedrich Hayek, an Austrian economist, in presenting his arguments on changes that can be brought about by P2P computing. It is conceivable that the P2P architecture can not only revolutionize how information is managed but also revolutionize our socio-economic and political institutions.

Lee [15] presented a survey of current users of P2P file sharing systems. The survey was focused on finding end-user perspectives on P2P systems. He surveyed the features

of P2P file sharing systems. The survey showed that the top five features are fee, speed, stability, reliability, and downloading technologies.

Kubiatowicz [13] discussed perils and problems associated with P2P computing. He drew a parallel with thermodynamic system design. He made an argument that the P2P system seemed chaotic as a thermodynamic system but it is manageable as long as we solve the issues relating to P2P systems such as latency, security, scalability, and availability. Those are the central issues in any P2P system, and we also discuss them in our design and implementation chapters.

Balakrishnan et al. [2] were working to develop algorithmic building blocks for P2P systems. They discussed their own and other researchers' algorithms for performing distributed lookup. P2P file sharing and storing systems' designers can use these algorithms with assurance about scalability of the lookup function compared to non-scaling approaches in some grass-roots P2P implementations. The implementation of these algorithms was made easier by the authors' pointers to reference implementation toolkits.

Schollmeier [20] laid out definitions of Peer-to-Peer networking, which we discussed in Chapter 1. He described three definitions and the motivation behind cataloging P2P networking into three broad groups.

In a Peer-to-Peer networks [6] article, Geoffrey gave an overview of P2P networking, describing P2P features, technologies, and challenges. The article gave examples including Napster, SETI@HOME, and Instant-messengers. The article contained many pointers to P2P sites for file sharing and P2P resource sites.

According to Geoffrey's article, Napster (www.napster.com) was developed by Shawn Fanning in January 1999 during his freshman year at Northeastern University. Napster is probably the most well known P2P system [6]. Fanning, a musician, developed the software to share his music with his friends on campus. Napster became a way to share files among a larger community. According to Schollmeier's P2P definition, Napster is

a hybrid P2P network.

At Napster's peak, it had 75 million subscribers sharing approximately 10,000 music files per second [6]. These statistics show that the Napster model is in some sense a success story in P2P communities.

In the article [17], Manoj, Anjana, and Andrew describe P2P networking as an alternative information sharing model to the current client-server model. The article makes a case for the advantages of P2P networking and presents challenges in implementing the P2P model. The advantages the article presents are load balancing, dynamic information repositories, redundancy, fault tolerance, content-based addressing, and improved searches.

The article describes that decentralization can also bring chaos to the model impacting on the quality of service (QoS). The strength of the decentralized model poses a difficulty in providing security for the system. The strength of joining and leaving the system at will also poses a possible connection problem in the network. The article also raises issues of traffic redistribution when one member has the information that everyone is requesting. The article summarizes, "More significantly, a private P2P network in which users exchange decision-making information can control membership, implement authentication schemes, and adopt standardized schemes to describe and classify content."

The article [3] by David Barkai describes P2P computing and its application. Barkai bases his P2P computing on the characteristics of P2P rather than a formal definition. The characteristics are resource sharing and direct communication among peers. The article categorizes P2P applications into three broad categories: Distributed Computing, Content Sharing, and Collaboration. Distributed Computing means sharing CPU resources. Content Sharing means sharing of files and storage, as in Napster. Collaboration means interaction of peers near real-time, for example, instant messaging and audio and visual communications.

The article presents architecture of an infrastructure for a P2P computing environment. It discusses a common P2P middleware architecture. First, the article presents basic requirements:

- Lack of trust: Access must be granted to members.
- Hardware heterogeneity: Hardware is of different types.
- Software heterogeneity: The infrastructure should be able to integrate any type of software.
- Network heterogeneity: The networks connecting peers differ in bandwidth and latency.
- Scale: Members can be as few as two or as many as millions.
- Intermittency: The systems and configuration are not fixed and static.
- Location: Location of peers does not matter.
- Autonomy: Peers do not have to know about each other to achieve a goal.
- Local policies: A sub-group of peers may create policies to protect information.
- Distance: Geographical distance is not an issue in the community.

The article explains how the basic requirements give rise to a common architecture for a P2P networking. It makes the case by discussing the application and issues with each category of P2P application.

The article raises technical challenges including Communication, Naming and Discovery, Availability, Security, and Resource Management. It describes those issues and possible solutions. The article concludes by recognizing that P2P networking technology is in an evolutionary phase. It acknowledges that companies such as Microsoft and Sun Microsystems have made progress, but there is still room for improvement and room for research and development.

2.2 Distributed Computing

Distributed computing has been a research topic since the birth of the computer. Early research and developments in distributed computing have been in modeling hardware rather than software. Computer engineers at major companies such as AT&T, Xerox, and IBM, and at research institutions developed models for hardware to achieve distributed computing. Those studies and experiences set the stage for the modeling of distributed software.

The "Search for Extra-Terrestrial Intelligence" (SETI@HOME, http://setiathome.ssl.berkeley.edu/) project collects data from different radio telescopes, and searches for evidence of Extra-Terrestrial Intelligence. Since an enormous volume of data is collected from telescopes, processing those data takes many years to complete, even for a fast super-computer. For this reason, SETI@HOME takes advantage of distributed computing to process the very large data set.

The SETI@HOME model is a 'massively' distributed computing model [11]. The problem is sliced into smaller tasks, and each task is given to a participating client. A server keeps track of participant clients and their work units. If a participant fails to deliver results, the task is sent to different participant. The client application is run as a screen saver such that the processing takes place only when the CPU is idle on the client computer.

Our observation of the SETI@HOME architecture and implementation is that the client software has memory of the data processed. The client application is able to pick up where it has previously paused. Similarly, the server is able to piece together results that it received from many clients that are processing various pieces of data at any time.

The SETI@HOME project collects data from each radio telescope and stores the collected data in high capacity tape drives. Then, the data is divided into 50 seconds of a 20 kHz signal, which is 0.25 MB of data. "On the receiving end a 0.25 MB chunk

will require 1.3 sec on an incoming T1 line of 190 kb/s, or 2.3 minutes on a 14.4 k baud line. Upon completion (typically after several days) of the data analysis for each chunk, a short message reporting candidate signals is presented to the customer and also returned to Big Science and to the University of Washington for post-processing" [21].

The article [28] by Waldo, Wyant, Wollrath, and Kendall from Sun Microsystems Laboratories sets a stage for distributed computing and its challenges. The authors describe distributed computing by comparing it to local computing. The article describes the differences and challenges in distributed computing by comparing the same issues in local computing. The article presents distributed computing more or less as remote procedure call (RPC) extended to the object-oriented paradigm. In distributed computing, nothing is known about a recipient other than that it supports a particular interface [28]. The article also raises fundamental issues pertaining to distributed computing regardless of hardware or software model:

- Latency: The difference in time between a local object invocation and the invocation of an operation on a remote object.
- Memory access: An issue with an access to a memory by a local processor or a remote processor. The article presents two choices: either all memory access must be controlled by the underlying system or the programmer must be aware of the different types of access.
- Partial failure: One of the sub-system fails to respond. Since there is no global state, no agents can determine if a component has failed and inform other components about the failure. This is a central reality of distributed computing.
- Concurrency: The distributed environment introduces truly asynchronous operations; thus giving rise to problem of concurrency. The article does not mention distributed computing specifically. However, it states that concurrency is either handled entirely or not handled at all. There is no partial solution to concurrency.

"One could take the position that the way an object deals with latency, memory access, partial failure, and concurrency control is really an aspect of the implementation of that object, and is best described as part of the "quality of service" provided by that implementation" [28]. The authors argue that in addressing the quality of service of a distributed system, issues of latency, memory access, partial failure, and concurrency are also addressed. The argument is that these issues need to be addressed in any implementation. By coding in a certain way, we can avoid partial failure, memory access problems, and concurrency problems. However, the authors also claim that robustness is not a function of implementations. Their position is the robustness is inherent to the architecture and design of the distributed computing environment.

2.3 Peer-to-Peer Distributed Computing

The paper [5] is a white paper/case study by Intel Corporation applying Peer-to-Peer computing to the financial industry, showing that it is possible to take advantage of a firm's technology investments by tapping into the unused resources of employees' high-performance PCs. The paper presents an architecture that takes advantage of existing computer resources of a company to solve a complex portfolio pricing problem, a risk hedge calculation, and a market and credit risk evaluation.

The type of distributed computing system presented in the paper is not a radical departure from the traditional client/server hub-and-spoke architecture. The proposed architecture uses the server to manage processing jobs, but relies on high-performance desktop PCs to provide the processing power. The paper argues that the hybrid network approach to distributed computing falls into the P2P network architecture because the PCs in the peer network can exchange information directly with one another.

The way the system works is that a large problem is divided into smaller work units. A server distributes work units to idle PCs. If the computation is interrupted, the server sends the work unit to another idle PC. The PCs process the given work unit

and send results back to the server. The server manages all the work and combines all results to produce the final product.

The architecture includes a managing server that sends tasks to a set of computers called 1st-tier peers. Computers in the 1st-tier request help from computers in the 2nd-tier. The 2nd-tier peers only communicate to one 1st-tier peer. Hence, there is a many to one relationship from a 2nd-tier peer to a 1st-tier peer. This is more like a cascading responsibility from managing server to peers. This architecture is closer to distributed computing architecture than to P2P distributed computing architecture.

"Ultimately, financial services firms can benefit from increased utilization of desktop resources, regardless of their size. Larger firms can increase their return on investment (ROI) from high-performance desktop PCs, and smaller firms can become increasingly competitive as the capital investment required for advanced computing techniques falls" [5]. The benefit of using this type of architecture is to harness idle CPU and bandwidth to solve CPU intensive problems.

2.4 Java Technology

The book by Freeman, Hupfer, and Arnold [7] lays out the foundation of distributed computing using JavaSpaces. It is a good theoretical book on JavaSpace and its application in distributed computing. It provides a very good theoretical background on the JavaSpace specification. However, the book provides little help in developing a fully functional sample program. It does not have a chapter on how to start a JavaSpace or even how to install a JavaSpace.

The book by Halter [9] is a very practical book for writing a JavaSpace application. It explains systematically building a simple JavaSpace program. Chapter 2 presents the details for installing and running a JavaSpace. From Chapter 3 onwards, Halter builds a simple application methodically. It starts with a simple program to find a JavaSpace.

The books Halter [9] and Freeman [7] are complimentary. Freeman [7] provides theories behind JavaSpace technology, while Halter [9] provides implementation-specific details.

The JavaSpace and the Jini specifications provide a detailed description of JavaSpace and Jini technologies. Those specifications are available at Sun Java site http://java.sun.com under documents and API section. The website http://www.jini.org also provides a portal to information on Jini.

2.5 Financial Portfolio Theory

In this section, we survey some relevant literature in finance.

The book by Brealey and Myers [4] covers corporate finance. It provides a background in valuation, risk analysis, financial decisions, and market efficiency. It describes market theories and hypotheses leading to market efficiency. It describes in detail financial decisions such as debt policy, dividend payout, and issuing securities.

The article by Kumar, Tan, and Ghosh [14] uses four different models to forecast foreign exchange rates. The rate forecasted is then passed through a rule-based trading system to generate buys and sells on the foreign currency. The four models are ANN, ANN with ARIMA, ANN with GARCH, and Random Walk Model (RWM). The article concluded that the ANN model was better in forecasting ability. When used with ARIMA and GARCH, ANN forecasting improved only slightly. The result showed that the Random Walk Model outperformed all other models in forecasting direction.

The book by Reilly and Brown [18] covers investment analysis and portfolio management. It discusses theories, techniques, and methodologies for investment analysis. It also presents many portfolio management techniques.

The technical document RiskMetrics [16] presents modeling techniques and theories in measuring market risks in a portfolio. The document presents the Value at Risk (VaR) method of measuring portfolio risk as well. It also discusses Monte Carlo simu-

lation of a portfolio value. The document is highly regarded in the financial industry.

The articles we discussed in this Chapter provide information on some work that has been completed in the field. The knowledge of previous research is a foundation for formulating the financial problem. In following Chapters, we describe the technology that we use to implement the design of the Octopus architecture.

3 Jini/RMI

Jini and RMI technologies are the foundations of the JavaSpace technology. The JavaSpace technology leverages Jini's distributed architecture and RMI's communication protocol. We present the basics of Jini and RMI in this chapter before we describe JavaSpace in the next chapter.

3.1 Jini

In this section, we provide a detailed overview of Sun Microsystems' Jini technology and cite additional relevant specifications. We use the Jini specifications [24] and [22] as out primary source literature.

Jini technology is a set of application programming interfaces (APIs) and services that provides a framework upon which a dynamic distributed system can be built and deployed. The Jini framework provides a technological foundation to extend a Java application interface from a single virtual machine to a network of virtual machines [23]. It exploits Java application interfaces and their characteristics to simplify construction of a distributed system. Its architecture adds mechanisms that allow components in a distributed system to easily share among entire networked members. Jini uses RMI to move objects in the network. The framework has built-in mechanisms for members to join and leave from a network dynamically at will. This is where the power of Jini architecture lies, by allowing members, who are providers and consumers of services, to dynamically join and leave the collective. The framework is designed as a collection of hardware devices and software components [22] within which members share services.

A service is an entity that can be consumed or published by a device, software, human user, or another service. The concept of 'Service' is central to Jini. A service is shared among members of a Jini system. For example, a service may be a simple computation program or a storage device. Generally, a consumer finds a service and a provider through a protocol. The consumer sends a service request to the provider.

Services may use other services to complete a given task. The Jini framework provides an infrastructure for defining, registering, and consuming services.

A Jini system consists of the following parts [22]:

- A set of components that provides an infrastructure for federated services in a distributed system.
- A programming model that supports and encourages the production of reliable distributed services.
- Services that can be made part of a federated Jini system and which offer functionality to any other member of the federation.

According to the Jini Architecture Specification [24], some of the goals are:

- Enabling users to share services and resources over a network.
- Providing users easy access to resources anywhere on the network while allowing the network location of the user to change
- Simplifying the task of building, maintaining, and altering a network of devices, software, and users.

At the heart of the Jini framework is a trio of protocols. They are *Discovery, Join,* and *Lookup.* These protocols are the base of the Jini framework for developing scalable distributed systems. As we briefly describe these protocols, it will be evident why a distributed system built using the Jini framework is scalable. The following subsections provide a brief description of each protocol.

3.1.1 Discovery Protocol

Discovery is the protocol to locate a 'Lookup Service' in the network [22]. Discovery of a Lookup Service is the first step in a Jini framework to become a member as a service provider or a consumer of a service. Therefore, it is a gateway to a distributed system.

Once a Lookup Service is located, a member can register as a service provider in the Lookup Service or scan the Lookup Service for available services and their providers.

There are three types of Discovery protocols. They are:

- Multicast request protocol: a service consumer performs multicasts to discover nearby Lookup Services.
- Multicast announcement protocol: a Lookup Service multicasts its locations to the world.
- Unicast protocol: a request-response protocol to discover Lookup Services. It uses the unicast transport protocol provided by the network.

3.1.2 Join Protocol

Join is the protocol to register a service to one or more Lookup Services in the network. A service provider uses the Discovery protocol to locate a Lookup Service. Then through the Join protocol, the provider registers its service to the Lookup Service to make the service available to all members. If this is a new service, then the Lookup Service broadcasts the availability of the new service.

3.1.3 Lookup Service Protocol

Consumers find and resolve services using a Lookup Service protocol. A Lookup Service provides a service to a consumer with an address of a service provider. The consumer uses the address to request services from the service provider. In other words, a Lookup Service links interfaces of service providers' functionality to the providers' implementation of the service.

A Lookup Service is a fundamental part of Jini infrastructure because it provides a registry of services available within a Jini system. It maintains a directory of service items that are instances of available services. When a service is registered with a Lookup Service, the service object contains a Java API for the service such that a consumer can invoke the API to request the service. A consumer queries the Lookup

Service to find a desired service. The query may return the API of more than one service provider. The consumer requests the service via the given API.

Here are examples of how a provider registers its service in a Lookup Service and how a consumer consumes a service in a Jini system using the protocols above.

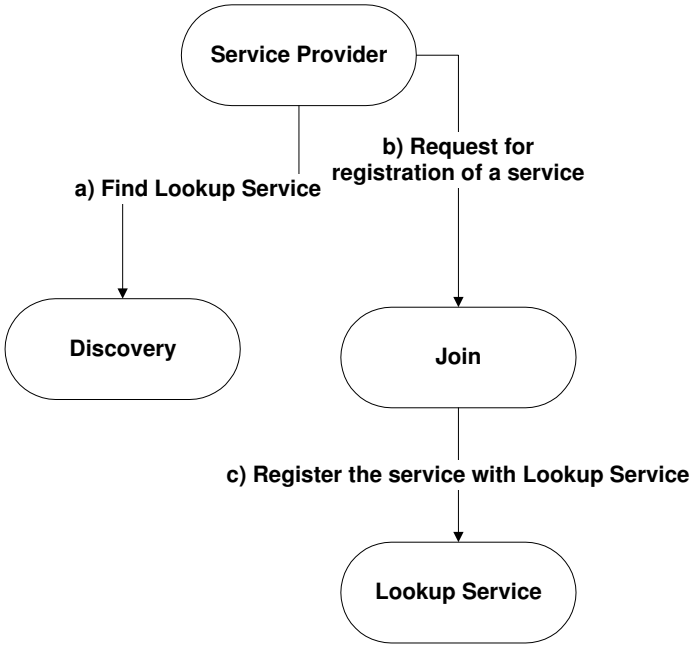

Figure 3: Service Provider registering a service in a Lookup Service

A service provider uses the Discovery and Join protocols to register a service. The registration process happens in the following steps shown in Figure 3:

- Find a Lookup Service using the Discovery protocol.
- Register a service or services using the Join protocol in the Lookup Service.

A service consumer uses the Discovery and Lookup Service protocols to find a service and a service provider. The process of finding and requesting services happens in the following steps:

- Find a Lookup Service using the Discover protocol.
- Find the desired service and service provider in the Lookup Service.
- Request the service from the service provider.

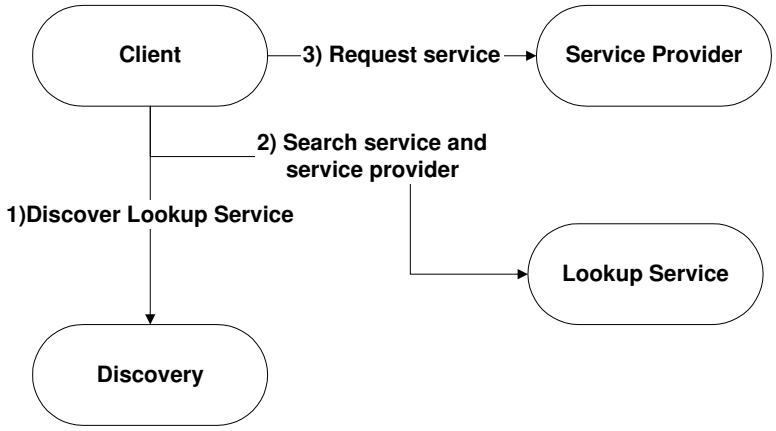

Figure 4: Client finding and requesting a service in the Jini environment

Figure 4 shows the above steps in finding a service in Lookup service, and then requesting for the service from a provider.

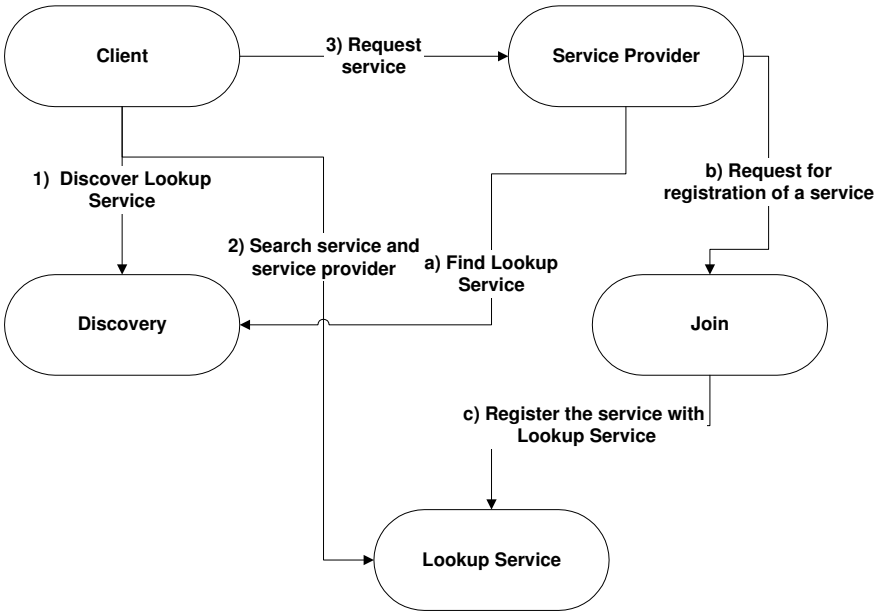

Figure 5: Jini Discovery, Join, and Lookup process

Figure 5 shows the process of Discovery, Join, and Lookup in a Jini environment by a consumer and a service provider.

For further reading, please refer to the Jini Architecture Specifications [23].

Jini is the foundation for JavaSpace. Therefore, a JavaSpace is developed to be a Jini service, taking advantage of the Jini framework for distributed computing [9]. Jini

provides a foundation upon which distributed computing systems can be developed using the JavaSpace specification. As a Jini service, a JavaSpace provides a repository for objects as a service to all members such that it provides a mechanism for sharing Java objects. We describe JavaSpace in detail in Chapter 4.

In the Jini Technology Starter Kit 2.0, there are a few additions to the Jini architecture[1]. The additions are mainly in application security and configuration. There are also improvements in programming models for service providers and service consumers. The new version adds support for pluggable invocation layer behavior and pluggable transport providers known as Jini Extensible Remote Invocation.

The security enhancements allows mutual authentication between service providers and service consumers, authorization, integrity checks (e.g. cryptographic checksums), and encryption. This improvement provides a framework for trust verification and a constraint-based remote invocation model. The constraint-based model allows applications to specify various security requirements on remote invocation. This provides the service provider with a medium to effectively shield itself from unwanted operation on its systems. This also allows the application to dynamically change the security requirement.

This version of Jini also changed the fundamentals of the service startup method. This version adds a *Configuration* programming model. This allows an application to be configured at deployment time. The **net.jini.config.Configuration** API provides a simple, uniform way to create configuration objects at deployment time. Jini provides **net.jini.config.ConfigurationFile** as a default implementation of **Configuration**. The configuration file is written in a subset of the syntax of the Java programming language such that Java objects can parse the file with ease. Please refer to the *net.jini.config package documentation* for an example and further information on the configuration.

[1]For further reading, please refer to the Jini Technology Starter Kit overview v2.0 that is included in the Jini Technology Starter Kit download.

Jini uses Remote Method Invocation (RMI) as a method of communication. The following section describes the RMI specification and its relevance to the paper.

3.2 Remote Method Invocation (RMI)

RMI is the means to move objects across the network using Jini [25]. RMI is a remote procedure call (RPC) mechanism in the J2EE framework. It is a communication vehicle among distributed objects. Since RMI is part of the grand Java specification, it gives a seamless communication method for writing a distributed application under Jini.

According to the RMI specification [26], general requirements for RMI to support distributed objects in the Java language are:

- Support seamless remote invocation on objects in different virtual machines.
- Support callbacks from servers to applets.
- Integrate the distributed object model into the Java programming language in a natural way while retaining most of the Java programming language's object semantics.
- Make differences between the distributed object model and local Java platform's object model tranparent.
- Make writing reliable distributed applications as simple as possible.
- Preserve the type-safety provided by the Java platform's runtime environment.
- Support various reference semantics for remote objects; for example non persistent references, persistent references, and lazy activation.
- Maintain the safe environment of the Java platform provided by security managers and class loaders.

We described the Jini and RMI technology that is the base technology for the JavaSpace technology. In the following chapter, we discuss the JavaSpace technology and how the JavaSpace technology extends Jini and RMI technological frameworks.

4 JavaSpace

In this chapter, we give a detailed overview of JavaSpace. We start by describing JavaSpace and then cite relevant features of the JavaSpace specifications [27]. We show the advantages of using Java technology to implement our P2P Distributed Computing model in the subsection 4.2. We use the JavaSpace specification [27] and the book [7] as the primary source for the literature.

4.1 Description of JavaSpace

A JavaSpace may be defined in several different ways depending on the viewpoints of an inquirer. Some of the perspectives from which we can look at JavaSpace are: a Jini Service, a mechanism for distributed communication, and a mechanism for object storage. In the Jini/RMI chapter, we have discussed that a JavaSpace is a Jini service. In this chapter, we present JavaSpace as a mechanism for distributed communication and object storage.

A JavaSpace is a network-accessible "shared-memory," which is a repository for Java objects [7]. Therefore, a JavaSpace provides a mechanism for object storage and distribution. Members can write to a JavaSpace as a persistent object storage, which in turn can be shared among members or processors. This sharing of objects allows members to use a JavaSpace as an indirect medium of communication. Hence, JavaSpace technology provides a fundamentally different computing model by decoupling members or processors from having to know each other as in client/server model. There can be one or many JavaSpaces. Figure 6 gives a visual representation of multiple JavaSpaces working with members or processors.

We present the JavaSpace technology and its key features in the following few paragraphs to show how the JavaSpace technology works.

The JavaSpace technology is heavily influenced by the concept of a tuple space, first described in 1982 in the Linda programming language [27]. Many active programs

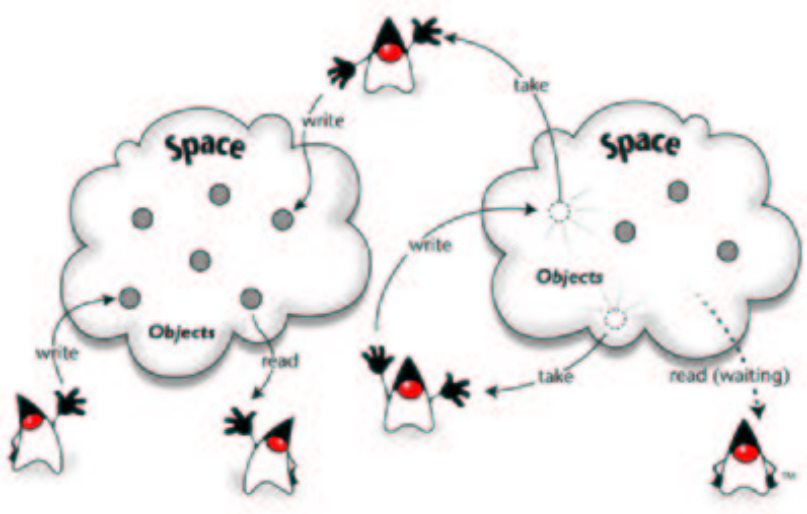

Figure 6: JavaSpaces [27]

can exist over physically dispersed processors, unaware of each other's existence, and yet still able to communicate with each other. They communicate by releasing data (a tuple) into tuple space. Programs read, write, and take tuples (entries) from a tuple space that are of interest to them. Similarly, JavaSpace is analogous to a tuple space where processors can write, read, and take Java objects. Members or processors communicate by exchanging objects through JavaSpaces rather than communicating directly.

A JavaSpace comes in two flavors, transient and persistent [27]. The transient nature of a JavaSpace is that when the JavaSpace goes away, objects in it also go away. However, in a persistent JavaSpace, objects in the space continue to exist even after the space is stopped and started. Hence, contents of a persistent JavaSpace are saved in disk storage. Once an object is written to a JavaSpace, that object exists in the space until a member explicitly removes it or the "lease" time for it expires, automatically removing it from the space. Since objects written in a JavaSpace are persistent, their life cycle is independent of their creators. Therefore, objects in JavaSpaces live even after their creator no longer exists. This persistence feature of JavaSpace technology removes dependency among members' life cycles. Members can communicate through

a JavaSpace using objects without having to know if a member is alive or not.

Members can write an object to a JavaSpace; but how can a member find the object or any object in the space? A member can locate objects in a JavaSpace using *associative lookup*, which is a way of finding objects by looking into the contents of an object without having to know an object's name and its creator.

JavaSpace technology provides a transaction model that guarantees an operation on a JavaSpace is atomic [27]. The transaction model has ACID properties that are 'Atomicity,' 'Consistency,' 'Isolation,' and 'Durability.' For further reading, please refer to the JavaSpaces specification [27] section 'Transactions and ACID Properties.' The transaction model supports operations over a single JavaSpace or over multiple JavaSpaces.

One of the key features of JavaSpace technology is that it allows exchanging executable content in an object [27]. Members cannot execute methods on an object in a JavaSpace. However, when a member reads an object, it receives a local copy of an object on which it can invoke any available method. This allows members to execute objects that they have never encountered before. This is often known as 'dynamic code downloading.'

We discussed writing objects into a JavaSpace. However, we have not defined objects that can be written to JavaSpaces. A standardization of objects is necessary for all members to understand and operate on objects in JavaSpaces. The JavaSpace technology uses Jini 'Entry' interfaces as a common interface for any object written to a JavaSpace. An Entry interface is defined in the Jini *Entry Specification* [24] and packaged in net.jini.core.entry.

A member can write an object to a JavaSpace if and only if the object implements the Entry interface. By restricting all objects to use a common interface, JavaSpace and Jini technologies standardize objects such that any member can easily process information in an object without having prior knowledge of the object. Having the

Entry interface implemented by all objects imposes the following rules:

- Subclasses of Entry must provide a public constructor without any arguments.
- All members of Entry must be defined as public.
- No primitive attributes are allowed in an Entry.
- When an Entry is serialized, each of its fields is serialized separately.

So far, we have discussed that a JavaSpace provides a medium through which members can communicate using Entry objects without having to know each other's existence. These Entry objects may have commands to solve problems, the definition of a problem, or method that perform some actions. The space does not execute methods on the Entries. It is simply a container for the Entries. As long as the "shared-memory" exists, the Entries in it also exist.

For example, a member can create an Entry that has a method to compute the factorial of a given number. It can then write this factorial Entry into a JavaSpace. Once the object is written into the space, it is available to members. Let us say a member reads the Entry out of the space to its local machine. Now the member can invoke the factorial method on the Entry to compute the factorial without having to know any information about factorials. Here the member is using the service provided by some other members. In this way, members can exchange tasks and request information without having to know who has the relevant information. The details of how the space works is discussed in Chapter 8, Implementation.

A JavaSpace provides a basis for the implementation of a cooperative, distributed architecture. Members write and read Entries from the space. A matching criterion is used to take Entries out of the space. Those Entries represent services, work in progress, states, and other information or data shared among the networked resources. Therefore, the JavaSpace technology is a very good candidate for developing distributed applications.

The following are a few JavaSpace specifications to illustrate why we are embracing this technology to implement our P2P Distributed Computing model and the advantages it provides.

JavaSpace specification JS.1.5 Goals and Requirements [27] states that the goals of the design of the JavaSpace technology are:

- Provide a platform for designing distributed computing systems that simplifies the design and implementation of those systems.
- The client side should have few classes, both to keep the client-side model simple and to make downloading of the client classes quick.
- The client side should have a small footprint, because it will run on computers with limited local memory.
- A variety of implementations should be possible, including relational database storage and object-oriented database storage.
- It should be possible to create a replicated JavaSpaces service.

According to the JavaSpace specification, the requirements for JavaSpaces application clients are:

- It must be possible to write a client purely in the Java programming language.
- Clients must be oblivious to the implementation details of the services. The same entries and templates must work in the same ways no matter which implementation is used.

JavaSpace specification JS.1.6 Dependencies [27] states the JavaSpace specification relies upon the following other specifications:

- Java Remote Method Invocation Specification [26].
- Java Object Serialization Specification [24].

- Jini Entry Specification [24].
- Jini Entry Utilities Specification [24].
- Jini Distributed Event Specification [24].
- Jini Distributed Leasing Specification [24].
- Jini Transaction Specification [24].

JavaSpace specification JS.2 Operations [27] states that there are four primary kinds of operations that can be invoked on a JavaSpace service. They are as follows:

- Write: write the given entry into a JavaSpace service.
- Read: read an entry from a JavaSpace service that matches the given template.
- Take: read an entry from a JavaSpace service that matches the given template, removing it from this space.
- Notify: notify a specified object when entries that match the given template are written into a JavaSpace service.

JavaSpace specification JS.3.2 [27] specifies Transactions and ACID (Atomicity, Consistency, Isolation, Durability) Properties. The following are ACID properties defined in JavaSpaces:

- Atomicity: all the operations grouped under a transaction occur or none of them does.
- Consistency: the completion of a transaction must leave the system in a consistent state. Consistency includes issues known only to humans, such as that an employee should always have a manager. The enforcement of consistency is outside of the transaction - a transaction is a tool to allow consistency guarantees, and not itself a guarantor of consistency.

- Isolation: on-going transactions should not affect each other. Any observer should be able to see other transactions executing in some sequential order (although different observers may see different orders).
- Durability: the results of a transaction should be as persistent as the entity on which the transaction commits.

The JavaSpace technology provides a framework to implement the Octopus architecture. Since the JavaSpace, Jini, and RMI technologies are well defined and support distributed computing architecture, we can leverage the existing framework to develop the implementation of the Octopus. We are taking advantage of the freely available technologies, which give us the ability to develop a prototype in a short period.

4.2 Benefit

The above description of JavaSpace and the synopsis of the specifications outline technological advantages in building a distributed system with the JavaSpace technology. The benefit of using JavaSpace is that the specification is well defined that supports loosely coupled protocols. This allows us to develop a system where there is no notion of server or master, and all members are equal.

Sun Microsystems has already implemented the JavaSpace specification. Therefore, instead of re-inventing the wheel, we use Sun's existing implementation of Java, Jini, and RMI. On top of that, API's are freely available through Sun. This provides a free tool to implement our architecture.

Since we are using Java technology, we are depending on the Java motto of "write once and run everywhere." This gives a new meaning to our architecture, as it does not depend on the type of hardware and CPU. It becomes a true distributed environment. The only requirement is a client application that runs on each participating computer.

We laid out the JavaSpace and Jini specifications. Let us explain the components of Octopus in detail in the following chapter.

5 Octopus: An Architecture for Peer-to-Peer Distributed Computing

As we have defined in section 1.2, distributed computing is about designing and building applications that are distributed across a network of computers to solve a single problem. A Peer-to-Peer (P2P) network is one way the computers may communicate. Taking these two concepts, we present in this chapter a P2P Distributed Computing environment named **Octopus**. The goal of Octopus is to solve a CPU intensive problem by using unused processing time on networked desktop computers. We define requirements for Octopus and its components in section 5.1. We explain the design process in section 5.2. We develop the architecture of Octopus in subsection 5.2.1. We discuss details of the implementation in Chapter 8.

5.1 Requirements

In this section, a set of high-level requirements for the Peer-to-Peer Distributed Computing model (Octopus) is laid out, followed by detailed use cases that capture workflow.

We give a brief definition of a Distributed Communication Space (DCS). A Distributed Communication Space is an area where members have access to read and write objects. The DCS is a means to communicate information among members in the community. We discuss details of DCS in Chapter 8

The high level requirements for developing a Peer-to-Peer computing architecture are:

- A tenet of developing the system is to exploit idle CPU time on networked computers.
- Membership in the community is voluntary. Members volunteer in publishing problems and acquiring solutions.

- Members can join and leave the community at any time without having any impact.
- Members publish a problem in a Distributed Communication Space (DCS) asking members that acquire problems from that DCS to help solve a problem.
- Members or acquirers get a problem from a DCS. They find solutions to the problem and publish their solutions to the DCS.
- Ability to transfer a large data set from a publisher to an acquirer of a problem.
- Ability to process problems that are CPU intensive.
- Ability to solve a large problem in a short time period.
- Ability to ask many members to solve a problem such that at least one solution is received.

In the above requirements, there is no constraint on a problem. The system should be able to handle any problem whether large or small as long as we can distribute the problem. For our exercise, we are taking a FP to illustrate our design of the P2P distributed system. In general, if a problem has a large data set, we divide the data into small units. We define such a unit of data as a **work unit**. A problem has two components: one data and the other model of the problem. A model is a set of well-defined instructions for processing the data to produce a solution. We define a **ticket** as a class containing both model and data of a problem or a solution to the problem. A ticket may contain either a reference to a model and data or a payload of model and data itself.

With the above requirements in mind, we describe two broad use cases as follows. Use Case 1 describes actions of a publisher, while Use Case 2 describes actions of an acquirer.

5.1.1 Use Case 1: Publisher

Context: A Publisher is a member of a volunteer community that has a problem which needs solutions. In this use case, we describe how a volunteer member can publish a problem. The publisher has to follow these steps to publish the problem into a DCS. We also describe how the publisher receives solutions to its problem.

1. The publisher has a well-defined problem with a model and a data set.
2. A ticket is generated containing either a reference to the file name of the model and its data, or the model and data may be sent as a payload.
3. The ticket is published into a DCS.
4. If the ticket contains references to model and data, provide the model and data set to acquirers of the problem
5. Periodically check whether a solution is available in the DCS.
6. Get solution tickets from the DCS.
7. If the ticket contains a reference to the solution, request the solution from the acquirer. Otherwise, unpack solutions from the ticket. Save the solutions to disk.
8. Analyze and decide if the result is acceptable.
9. If the result is acceptable, remove the problem from the DCS. Otherwise, leave it in the DCS for another acquirer.

5.1.2 Use Case 2: Acquirer

Context: An acquirer is a member of the volunteer community who is waiting for a problem. In this use case, we describe how an acquirer gets a problem from the DCS, and how it publishes the solutions to the DCS. The following are the steps an acquirer takes to complete its tasks.

1. Periodically check the DCS for a ticket containing a problem.

2. Take a ticket from the DCS.

   (a) If the ticket contains references to a model and data, request the model and data sets from the publisher.

   (b) Save the data and model sets on the local drive.

   (c) Start processing the work unit as a low priority process.

   (d) Once processing is complete, generate a solution ticket with the model and solution data set or a reference to the model and solution data.

   (e) Publish the ticket to the DCS.

   (f) If the ticket is published with a reference, then when the publisher requests the solution, send it to the requestor.

3. Purge the data, model, and results from the local disk to make resources available for use.

5.1.3 Flow Diagram

We present a flow diagram based on the above use cases. Figure 7 represents the flow of information and the sequence of events in the system. The figure shows the flow of above use cases.

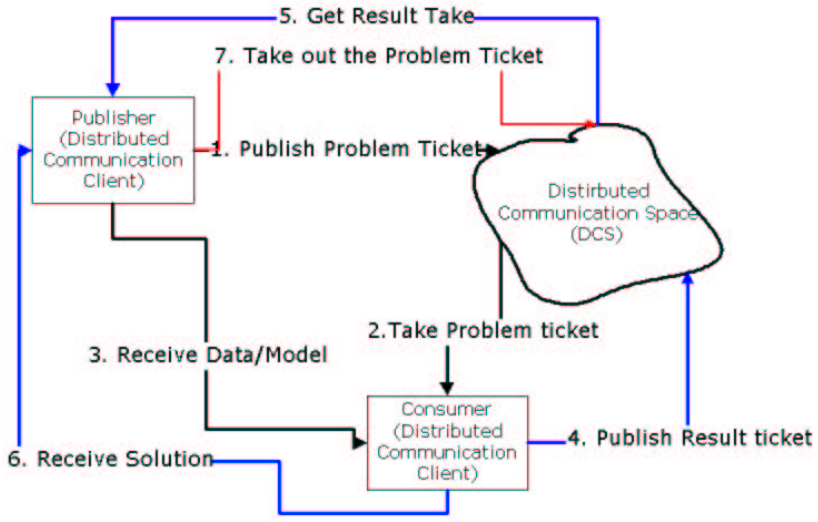

Figure 7: Flow of information

We have laid out requirements for a system and provided use cases and a process flow. We also require that system be scalable and fault tolerant. With these requirements, we design a system in the following section.

5.2 Design

In this section, we develop an architecture that meets the requirements outlined in the section 5.1. Then, we fully develop a technical solution for the architecture. First, we give a brief overview in this section. In the Architecture section 5.2.1, we develop a model that provides a response to the requirements. In the Design section 5.2.2, we lay out a detailed design specification that meets the requirements.

We develop an architecture using a Hybrid Peer-to-Peer networking and Distributed Computing model. Taking the architecture as a guide, we design and implement the Octopus system to demonstrate the validity of Hybrid Peer-to-Peer Distributed Computing. We use FP problems to demonstrate an implementation of the architecture. The implementation is described in Chapter 8.

We architect the Peer-to-Peer Distributed Computing system to be independent of the problems to be solved. Octopus should work not only for a FP problem, but also for any well-defined problem. 'Well-defined problem' means that a member who volunteers to solve the problem has sufficient information to produce a solution or solutions. A FP discussed in detail in section 6.1 is an example of a well-defined problem.

5.2.1 Architecture

In this section, we give details of the architecture of our P2P Distributed Computing network environment. Our goal in this section is to use the concepts of P2P networking and Distributed Computing to develop architecture for Octopus. Figure 8 shows the high-level the architecture of Octopus.

In Figure 8, V_1 though V_8 are member computers. They have the Distributed Communication Client (DCC), a client program, installed locally. The members publish

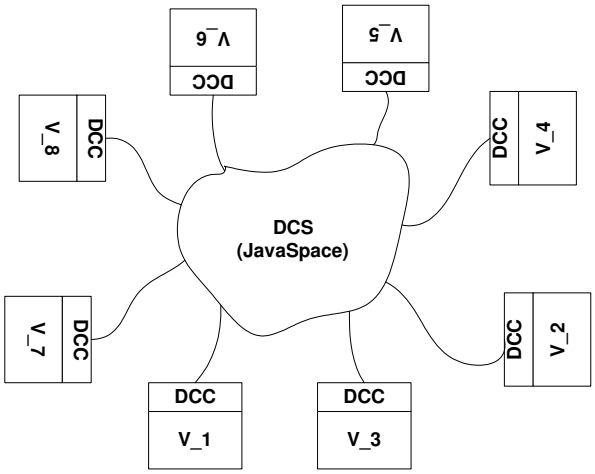

Figure 8: Octopus: High-level architecture

problems in the Distributed Communication Space (DCS), the JavaSpace, using the client program, DCC. Similarly, members use the DCC to acquire solutions from the DCS. In this environment, members do not know about each other. The only knowledge they have is about the DCS. Members neither know who is asking for solution to a problem nor know who is solving the problem. The DCC, a communication client, is a P2P networking application, while the DCS is a JavaSpace in the Octopus system.

For example, a publisher has a well-defined problem, a FP. It wishes to publish the FP to find a solution or solutions. The publisher instructs the DCC, a client program, to build a ticket describing the model and the data for the FP. Then the DCC publishes a ticket in the DCS. The publisher waits for a solution or solutions. It periodically checks in DCS for any reported solutions. The received solutions are processed and presented to the user. These are the characteristics of a publisher. Figure 9 describes the functional architecture of a publisher.

Similarly, an acquirer, a member who wishes to work on a problem, acquires a FP ticket from DCS. The acquirer invokes processing using unused CPU resources. Once the processing is complete, the acquirer builds a ticket containing the completed work, which is published in DCS for consumption by the publisher. The solution is provided to the publisher. Then the resources are released by deleting the data, the model, and

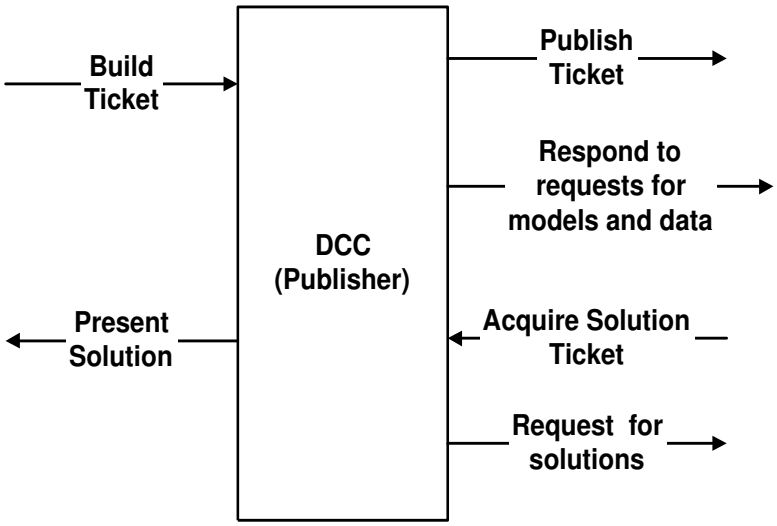

Figure 9: Architecture of DCC publisher

the solution files from the local disk. Figure 10 describes the functional architecture of an acquirer.

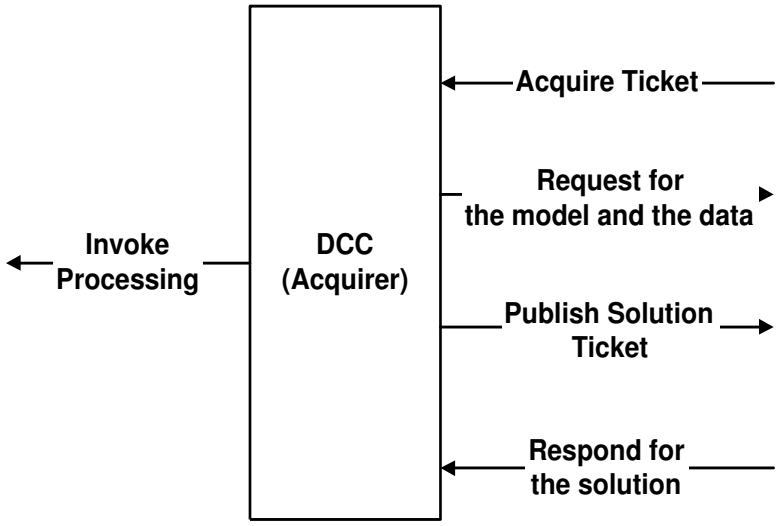

Figure 10: Architecture of DCC acquirer

Based on the requirements and functions of publishers and acquirers, the functional requirements for designing the Distributed Client Communication (DCC) are:

1. Find DCS.

2. Create and decipher tickets.

3. Read and write problem tickets from and to DCS.

4. Read and write solution tickets from and to DCS.

5. Respond to requests for the model and data.

6. Respond to requests for the solution.

The architecture of DCC should support publication of problem and solution tickets in DCS as well as acquiring of problem and solution tickets from DCS. We describe the details of the design of DCC in subsection 5.2.2.

We now discuss the architecture of a DCS. So far, we have said that a DCS is a JavaSpace. The requirement for a DCS is that it should be accessible by members in a networked environment such as the Internet or a private network. The architecture of the DCS needs to support members joining and leaving the community without any impact on the system. If a member decides to join in either publishing or acquiring a problem to solve, the member needs to install DCC and then find a DCS. The architecture also needs to support members leaving the network at their convenience. This means if a member wants to leave the community, all it has to do is to cease to communicate with the DCS.

The architectural question is how can we design DCS to handle the requirements of publishing and acquiring of problems and results and of allowing members to join and leave the community at their leisure. Figure 11 shows the architecture of DCS.

As shown in Figure 11, a DCS has two channels, one for problems and the other for solutions. The channel for problems is called the **Work unit Publication Channel (WPC)**, while the channel for solutions is called the **Solution Publication Channel (SPC).**

A publisher and an acquirer use WPC and SPC to communicate problems and results. A publisher publishes problem tickets on WPC. An acquirer looks for a problem ticket in WPC. Once the acquirer completes processing of a problem, it publishes solution tickets on SPC. The publisher searches for solution tickets in SPC. We describe

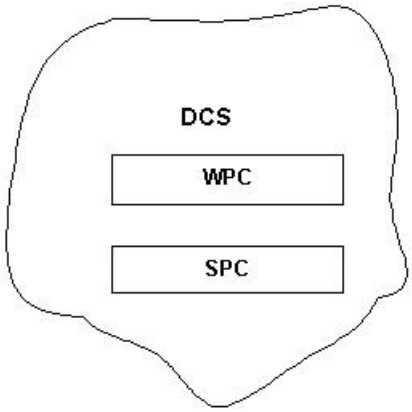

Figure 11: DCS architecture

the detailed design of these channels in the Design section 5.2.2.

We have presented the architecture of Octopus at a high level in this section. In the following section, we discuss our architecture in detail.

5.2.2 Design

In this section, first we describe how a DCS and its channels are designed. Then, we describe the design of DCC.

A DCS has two channels, WPC and SPC, as shown in Figure 11. The channels are designed as queues. The design of each queue is such that members using DCC are able to do such operations as en-queue, read, and de-queue. The design of a SPC is simpler than the design of WPC. We present the design of SPC followed by the design of WPC.

We design a SPC as a simple queue-like structure. The queue-like structure is a placeholder for solution tickets. The ticket may contain a digital signature to allow only the publisher to remove the ticket from the queue. Publishers are allowed to search through the queue to de-queue tickets that they own.

A WPC needs to support more than one acquirer access the same problem ticket. Only publishers are allowed remove their own problem ticket from the queue. We support those requirements by designing the WPC as a **Circular First In First Out**

(CFIFO) queue. A CFIFO queue is a FIFO queue with a caveat that when a ticket is removed from the queue, a copy is re-queued. For example, the WPC has a, b, and c elements in order. When an acquirer removes the first element a from the WPC for processing, the element a is re-queued immediately. Now, the WPC has elements b, c, and a in order. Figure 12 gives a visual representation of a CFIFO queue.

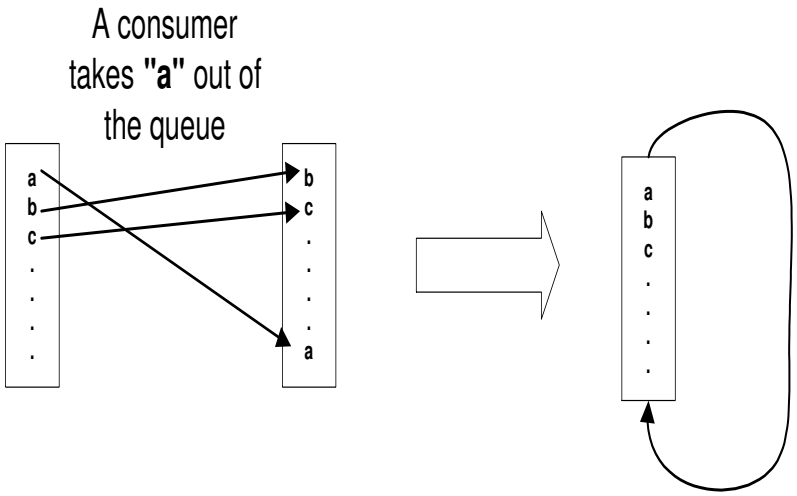

Figure 12: CFIFO (Circular First In First Out) queue

By design, a CFIFO queue allows more than one acquirer to work on the same problem. Therefore, if there is only one ticket on the queue, then all acquirers receive the same ticket. More tickets on the queue mean that acquirers will receive different tickets. Since tickets are re-queued, one ticket may be picked up by more than one acquirer. This is the strength of the design of the WPC. We are allowing more than one member to work on a problem by design. This gives publishers the opportunity to ask many acquirers to work on same problem until a solution is obtained.

The architecture of the CFIFO queue gives rise to a possibility that a problem may potentially live in the queue indefinitely. Therefore, it is the responsibility of a publisher of a problem to remove the problem from the WPC. In fact, only the publisher of the problem is allowed to remove the problem from the queue. To allow only publishers to remove the ticket from the queue, tickets may contain a digital signature. This does

not entirely solve the problem of a ticket living in the WPC indefinitely because a publisher can disappear. Therefore, we solve the problem of a ticket being in the WPC indefinitely by having an expiration time on tickets. This way, either a publisher takes the ticket out or the ticket expires.

We presented the design of the DCS. Now, we discuss the design of the DCC.

Distributed Communication Client (DCC) is a client application. We design DCC as an application that allows finding a DCS and communicating tickets to and from it. The design of DCC is such that a member can be either a 'Publisher' or an 'Acquirer.' Figure 14 describes the design of DCC.

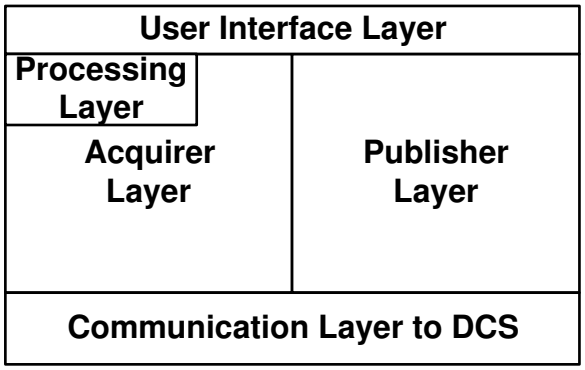

Figure 13: Architectural design of DCC

We design DCC to have four layers. The communication layer to DCS is the base layer for the publisher and acquirer layers. It provides the utilities and APIs to find, connect, and make requests to DCS.

The user interface is a top layer, which allows users to interact with the DCC. Users issue a publish and an acquire command through this layer. The publisher layer implements requirements to be a publisher. The acquirer layer implements requirements to be an acquirer. It has a processing sub-layer that allows it to interface with external systems to process problems.

All the layers are implemented using Java. We describe the details of their implementation in Chapter 8.

Figure 14 describes the design of Octopus. It captures the interaction between the DCC and DCS layers.

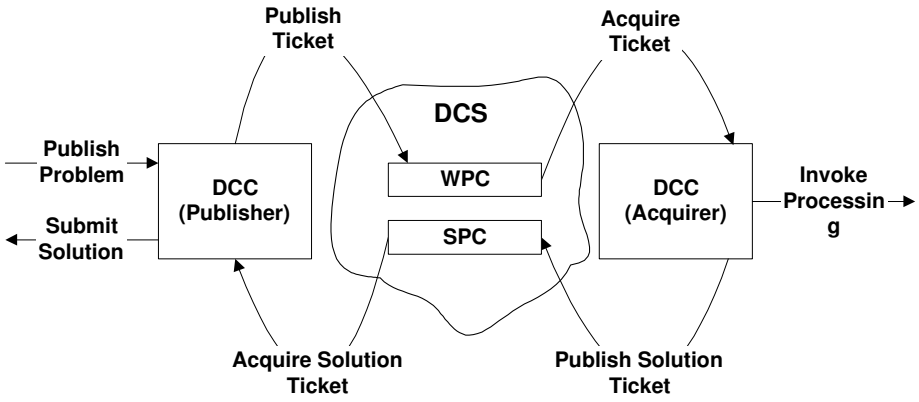

Figure 14: Architectural design of Octopus

5.3 Technical Issues

Though the distributed application has many appealing characteristics, there are many challenges that the system faces compared to local computing. The major issues in distributed computing are latency, memory access, partial failure, concurrency, and synchronization [28].

The Octopus architecture is for a time-consuming problem. It is best for those problems that require complex computation on a large data set although by design any type of problems can be solved. The time lag in communicating between two computers in a network is called **latency**. Latency in Octopus is the time lag in communicating between DCC and DCS. Our choice of the problem is such that the ratio between the latency and computing time is very low. That is, the transfer of data may take 20 seconds, but the computing time may take 20 hours. Even though it takes a long time to get the data, the latency is not an issue in the architecture for problems with this very low ratio of latency to computing time. Besides, we are interested in solving a large problem in a reasonable time. Therefore, our design takes latency into account.

The architecture of Octopus does not require an acquirer of a problem to respond

with a solution. Once an acquirer gets a ticket, it is totally up to the acquirer to decide if it wants to process the work unit. There are numerous reasons why the acquirer may not reply with a solution. For example, a system failure or a hardware failure may cause the volunteer to abandon processing, or a network failure may prevent the volunteer from communicating to DCS. This is defined as **partial failure.** In this environment, partial failure has minimal consequences because there is a high probability that another volunteer will respond.

The architecture supports partial failure. Partial failure occurs when a peer does not send a solution to DCS. Members are communicating problems and solutions with DCS independent of each other's behavior. DCS is designed such that one or many members can work on the same problem. Furthermore, the architecture is decoupled in the sense that none of the members is required to answer back with results. In many cases, a publisher receives more than one response. Therefore, publishers are required to handle reports of multiple solutions. We view this as strength of the architecture.

The architecture handles **concurrency** through JavaSpace's implementation of transaction processing. Concurrency arises when multiple peers try to access the same resource. JavaSpace supports transactional processing, therefore allowing concurrent processing. For further reading on concurrency, please refer to [7].

The architecture is **scalable** as it allows members to join and leave the community with ease. If the problem is large, then we request volunteers to join in solving the problem. Similarly, if there are many unused resources, then the members will solve the same problems. The architecture addresses the scalability issue by design.

The system is **fault tolerant** with high availability because of the redundant nature of the architecture. If a member goes down or does not report solutions, the work will be picked by another member waiting to get a work unit. If a publisher goes down for a period, the tickets are still in the DCS. Therefore, when the publisher comes back, it will be able to process the solution tickets. Here we are assuming that the publisher

comes back before tickets expire.

6 Problem Definition

We laid out the architecture and environment for P2P Distributed Computing in the preceding chapters. In this Chapter, we lay a foundation for financial theories and Artificial Neural Networks (ANN). We explain market theories and instrument valuations. Here the term 'instrument' means financial securities traded in an open market. We call the financial problem we use here FP, 'Financial Problem.'

The FP problems we use are finding stock quotes and a Monte Carlo simulation of value of a simple equity portfolio consisting of four equity instruments.

We develop a portfolio and valuation theory of a security in section 6.1.

6.1 Financial Theory

In this section, we give an introduction to finance as a foundation for understanding the FP problems. We explain relevant theories that provide a basic understanding of financial market principles and the valuation of financial instruments. In this section, we use mathematical notations that are primarily used by economists. This description follows that of Brealey and Myers [4].

A portfolio consists of one or more financial instruments. Each instrument in the portfolio is called a holding of that instrument. The market value of a portfolio V at time t is the weighted sum of holdings at price P,

$$V_t = \sum_{n=1}^{N} w_{nt} P_{nt},$$

where w_{nt} is the weight in the portfolio of the instrument n at time t. We can think of w_{nt} as number of shares of instrument n at time t being held in the portfolio V. For example, suppose **YAHOO** is the 10^{th} instrument in the portfolio V. Thus, N is the total number of instruments held in the portfolio V. P_{nt} is the price of the instrument n at time t. In this example, $P_{nt} = \$1.50.$, and $n = 10$, which indicates YAHOO stock. The weight of YAHOO stock is $w_{10} = 5000$ on day $t = 01/02/2003$. The weight w_n changes as we trade, while the price of instrument P_n changes due to the market.

For example, suppose we have a portfolio A that consists of three instruments. They are Sun Microsystems with a weight of 500 shares, Oracle with a weight of 700 shares, and Cisco with a weight of 300 shares. On Jan 3, 2003, at time t, Sun's closing price is \$3.55, Oracle closed at \$11.56, and Cisco closed at \$13.91. The market value of A for t = Jan 3, 2003 is given by

$$V_{01/03/2003} = (500 * \$3.55) + (700 * \$11.56) + (300 * \$13.91) = \$14,040.00$$

where, $n = 1, 2, 3$, $N = 3$, and t = Jan 3, 2003.

A difference in market value of a portfolio at time t compared to time $t - 1$ = Jan 2, 2003, is the amount of money the portfolio made, which can be either positive or negative. For example, on Jan 2, 2003 the closing prices are Sun \$3.38, Oracle \$11.21, and Cisco \$13.64. The market value of the portfolio on Jan 2, 2003 for the same weights is \$13,629.00. Therefore the difference in market value of the portfolio A between Jan 2 and Jan 3 of 2003 is +\$411.00. This means the portfolio A has gained \$411.00 on Jan 3, 2003 due to changes in price.

The expected rate of return for a portfolio of instruments is the weighted average of the expected rates of return for the individual instruments in the portfolio,

$$E(R_p) = \sum_{n=1}^{N} w_n E(R_n),$$

where R_p is the return on the portfolio; R_n is return on instrument n in the portfolio, w_n is the weight of instrument n in the portfolio, and N is the total number of instruments in the portfolio.

The risk of a portfolio is the measure of possible fluctuation of the portfolio value. The rate of return usually fluctuates because the change in a price of an instrument causes the rate of return to change. In mathematical terms, the risk is the standard deviation of the market value of a portfolio. The standard deviation is computed as the sum of weighted variance and covariance of each instrument in the portfolio,

$$a_p = \sqrt{\sum_{i=1}^{n} w_i^2 a_i^2 + \sum_{i=1}^{n} \sum_{j=1}^{n} w_i w_j Cov_{ij}},$$

where a_p is the standard deviation (the risk) of the portfolio, a_i is the standard deviation of Ith instrument, w_i is the weight of the ith instrument in the portfolio, and Cov_{ij} is the covariance between the rates of return for instruments i and j, that is $Cov_{ij} = r_{ij}a_i a_j$.

We have laid out a basic foundation of the financial theory. We move on to the pricing of instruments. To understand the pricing mechanism, we need to know how a financial market operates and its assumptions.

A price of a good is determined by the supply and demand for the good [4]. This basic economic theory applies to financial instruments as well. Investors are willing to invest in a financial instrument that they believe will make money, that is, attain a higher price. An investor's willingness to pay for a unit share of a company's stock is its share price. A share price is determined by how much money a company is likely to make in the future and the supply and demand of the instrument in the open market. Therefore, a price is a measure of the stream of cash flow at today's value, discounted by the opportunity cost of the capital. The opportunity cost of capital is the rate of return that one could make by investing the money in another instrument. For example, suppose Shirish has $\$1,000.00$. He has a few alternative investments for the money. He can go fly-fishing and spend it. Alternatively, he can buy Sun Microsystems's shares, hoping the company will make money in the future and thus expecting the share price will go up. Or finally, he can just put the money in a savings bond. If Shirish decided to buy Sun Microsystem's shares for $1000.00, then the alternative investments, buying the savings bond or going fly-fishing represent the opportunity cost of $\$1,000.00$ of capital. We know the interest rate on the savings bond. This rate is often called the opportunity cost of capital. Hence, if we know how much Sun Microsystem can make in the future, we can use this interest rate to discount Sun Microsystems' future cash flow to obtain the present value of that future income today.

In determining the price, a financial market is assumed efficient. An efficient capital

market is one in which security prices adjust rapidly to the release of new information. Therefore, the current prices of securities reflect all publicly available information about the security [4].

The efficient market assumes that (refer [4] for details)

- A large number of competing, profit-maximizing participants analyze and value securities independent of each other.
- New information regarding securities comes to the market in a random fashion, and the timing of one announcement is generally independent of others.
- Competing investors attempt to adjust security prices rapidly to reflect the effect of new information.

"Investors do not buy a stock for its unique qualities; they buy it because it because it offers the prospect of a fair return for its risk" [4]. This means that all the financial instruments are perfect substitutes. Hence, the demand for a stock is highly elastic. For a perfect substitute, a consumer is indifferent in consuming either one of the products. The elasticity of demand is a measure of the percentage change in the quantity demanded for any stock when the price increases by a one percent. For example, if the IBM stock price increases but the Microsoft price is comparatively lower and the prospective return relative to risk is same, investor will switch from holding IBM to holding Microsoft.

When the market is efficient, there can be no "arbitrage" opportunity. In its simplest form, arbitrage means taking simultaneous positions in different assets to guarantee a riskless profit higher than the riskless return given by U.S. Treasury bills. If such profit potential exists, we say that there is an arbitrage opportunity. In efficient financial markets, current prices fully reflect all available information and are consistent with the risk involved. This means that no investor has more information than others. When new information is released for an instrument, the price of that instrument

reflects the new information immediately so that there is no arbitrage opportunity. There are two kinds of arbitrage opportunities.

The first kind of arbitrage opportunity is to make investments such that there is no current net commitment but there is the expectation of a positive profit. For example, we can short-sell a stock and use the income to buy call options written on the same security. Here, the investor has a long position in call options and a short position in the underlying security. Such investment should not yield any excess profit after commission and fees are deducted. However, if the investment does yield a positive profit, we say that there is an arbitrage opportunity of the first kind.

The second kind of arbitrage opportunity is to make an investment with a negative net commitment at present, but non-negative future profits. We can give a similar example as above to illustrate the opportunity.

We say that the price of a security is at a 'fair' level, or that the security is correctly priced, if there are no arbitrage opportunities of the first or second kinds at those prices.

With the above information on how prices are set in the market, we can define the following model for the price of a security,

$$P_n = f(X_{nt}),$$

where P_n is the price of security n, X_{nt} is an information set relating to security n at time t. A set of information may consist of economic indicators, such as interest rate, inflation, or unemployment in the sector. "Sector" is an economic term for section of the economy and the market where products and services are alike. The information set may also contain company specific information such as market share, earnings per share, total assets, market capitalization, price per earnings, bid-ask spread, etc. The information set varies depending upon the sector and industry to which the company belongs. For example, the set of information used to predict the price of IBM is different from the set of information used to predict the price of Starbucks. This is because IBM is a technology company whose products and services are totally different

from Starbucks, which is in the restaurant business.

In the following section, we discuss Monte Carlo simulation of a portfolio.

6.2 Monte Carlo Simulation

We use a Monte Carlo approach to do a portfolio value simulation. We take following approach for Monte Carlo simulation.

1. As we have described above, we create a financial portfolio, TECH, with a number of instruments, for example, YAHOO, SUN, BEA, and IBM.

2. We give a weight for each instrument in the portfolio. For our example, we choose the following weights for each instrument. They are 500 common stock shares of YAHOO, 300 common stock shares of SUN, 200 common stock shares for BEA, and 400 common stock shares of IBM in our portfolio.

3. We use the portfolio valuation equation 6.1 to evaluate our portfolio at a given price. In our case, we use mean price over the period 3/30/2001 and 3/19/2004 daily prices. For example,

$$V_{\text{TECH}} = 500 \times P_Y + 300 \times P_S + 200 \times P_B + 400 \times P_I,$$

   where V_{TECH} is the value of the portfolio TECH, P_Y is YAHOO's price, P_S is SUN's price, P_B is BEA's price, and P_I is IBM's price.

4. We simulate our portfolio, TECH, value with instrument prices computed as follows:

$$P = \mu_P + (1.96 \times \sigma_P \times \text{Random}(0..1))$$

   where P is a price of an instrument, μ_P is mean of the price over the time period 3/30/2001 to 3/19/2004, 1.96 gives the 95% confidence interval in statistical terms, σ_P is the standard deviation of the prices over the time period 3/30/2001 and 3/19/2004, and a random number is generated between 0 and 1. This gives

a price of an instrument within 95% confidence internal. Each random number gives a new possible price, and thus a new portfolio value. With each new price for all instruments, we evaluate the value of portfolio. This allows us to simulate the value of portfolio with the induced changes in prices of instruments. Therefore, the model with price shock can be expressed as follows:

$$\begin{aligned} V_{\text{TECH}} &= 500(\mu_{yp} + 1.96 \times \sigma_{yp} \times Random(0..1)) \\ &\quad + 300(\mu_{sp} + 1.96 \times \sigma_{sp} \times Random(0..1)) \\ &\quad + 200(\mu_{bp} + 1.96 \times \sigma_{bp} \times Random(0..1)) \\ &\quad + 400(\mu_{ip} + 1.96 \times \sigma_{ip} \times Random(0..1)), \end{aligned}$$

where V_{TECH} is the value of the portfolio TECH, σ is the mean price, and σ is the standard deviation of an instrument over a period of time, μ_{yp} is YAHOO's mean price, μ_{sp} is SUN's mean price, μ_{bp} is BEA's mean price, μ_{ip} is IBM's mean price, and 1.96 represents a 95% confidence interval on the price of the instrument. σ_{yp} is standard deviation of the YAHOO price, σ_{sp} is standard deviation of the SUN price, σ_{ip} is standard deviation of the IBM price, and σ_{bp} is standard deviation of the BEA price.

5. The following are the means and standards deviation of instrument prices.
   YAHOO mean price, $\mu_{yp} = 22.8572$ and standard deviation, $\sigma_{yp} = 11.0897$.
   SUN mean price, $\mu_{sp} = 38.5232$ and standard deviation, $\sigma_{sp} = 6.7704$.
   BEA mean price, $\mu_{bp} = 17.0842$ and standard deviation, $\sigma_{bp} = 3.3128$.
   IBM, mean price, $\mu_{ip} = 91.7290$ and standard deviation, $\sigma_{ip} = 14.8172$.

6. We presented the simulated value of the portfolio with mean, median, and standard deviation statistics of the portfolio value in appendix A.

6.3 Stock Quote

In this section, we describe the problem finding a stock quote. A stock quote consist of information on a specified stock such as bid and ask price, high and low volume traded, the time of the quote and so on. We request a stock quote from the YAHOO Financial site. YAHOO provides a method to query their financial data. We provide YAHOO with valid ticker symbol, and the YAHOO provides real-time stock data in a predefined format. We parsed and present the data for use.

When the publisher chooses a stock quote, it chooses a ticker symbol that is included in the work unit. An acquirer requests stock quote data from YAHOO. The data from YAHOO is parsed to create a solution ticket.

We chose two distinct problems for our demonstration. With these problems, we demonstrate that the Octopus architecture is independent of problem types. A Monte Carlo simulation is a suitable type of problem for the Octopus. Problems that require intensive CPU power to solve are the best candidates for the Octopus. With stock quote problem, we demonstrate that the Octopus is also architected in solving a simple request such as finding a stock quote from a service provider in the network.

In the following chapters, we discuss risk-benefit of adopting the Octopus system and walk through the process of implementing the Octopus architecture and the problem that we have presented in this chapter.

7 Benefit and Risk of Using Octopus-Style Distributed Computing

There are benefits and inherent risks of using any networked computer system. In this Chapter, we explore the benefits of the Octopus architecture and discuss risks that the system possesses. We provide strategies to minimize each risk we present.

During system testing we present quantitative measurements (where possible) to gather empirical test results. Our approach to quantitative measurements follows standard benchmarking techniques. Specifically, we follow methodologies described by Hennessy and Patterson in [10] in conjunction with techniques defined by *Standard Performance Evaluation Corporation* (http://www.spec.org.)

In this Chapter, we develop a scalability metric. We present the findings and compare the outcome with our predictions in appendix B.

The following section describes benefits in using Octopus.

7.1 Benefits

In this section, we describe the benefits of using the Octopus architecture, which include

1. Solve a large-scale problem
2. Capitalize unutilized resources
3. Be independent of problem type
4. Be Scalable
5. Be fault tolerant and available

In the following subsections, we describe each of the benefits listed above.

7.1.1 Solve a Large-Scale Problem

The Octopus architecture makes it possible to solve large-scale problems that may not be solvable with a single very fast computer. A large-scale problem consumes a lot of computing resources.

A problem may qualify for a large-scale because it has a very large data set. Processing that data set may require enormous computing resources. For example, finding an extra-terrestrial signal in data recorded from telescopes, the SETI@HOME project requires processing a large data collected from radio telescopes located worldwide. The Octopus system allows us to distribute large-scale problems to a number of volunteer computers. It is possible to sub-divide the given problem such that we can work on a smaller problem and its solution. Smaller problems, and their solutions, can be processed to find a solution or solutions to the larger problem. Therefore, the P2P distributed computing concept with the computationally divisible problem allows us to solve a large-scale problem.

7.1.2 Capitalize Unutilized Resources

In this section, we discuss capitalizing unused computer resources by using the Octopus system.

We can obtain more from our capital investment than is current practice by using a distributed system - Octopus. Specifically, we propose to use idle computing resources to distribute and solve large-scale problems. Examples of idle resources are predominantly CPU, memory, and disk space. The architecture of Octopus is designed to use these resources to support solving problems. The following example illustrates our assertion as it pertains to Octopus.

Let us propose a hypothetical company called Foo Financial Inc. whose operating assumptions include

- The company has 1000 employees.
- The company's hours of operations are 8 am to 6 pm, Monday through Friday.
- Average working hours per employee per day is 10.
- Each employee has a computer. For simplification, we assume all computers are equivalent to a DELL Optiplex GX270 in CPU, hard disk, and memory types,

though computers participating in Octopus can be of any type and operating system. This assumption is merely to get a rough estimate of benefit.

- Computers are left powered on all the time.
- The company has a well organized Information Technology group which has a computer support team that supports computer operations including software and hardware maintenance.
- Computer maintenance hours are 3 am to 5 am, Monday through Sunday.

Foo has 1000 computers, which are mostly busy during the hours of operation 8 am to 6 pm, and idle for the remainder of the day and on the weekends. Users use computers for 10 hours, and the Information Technology (IT) group performs 2 hours of maintenance giving at least 12 unused hours a day. Therefore, each computer is used as much as 60 hours a week and idle for at least 84 hours a week.

During 2 hours for the system maintenance window, the technical support group pushes virus updates, new software, and any necessary software updates. The technical support team is able to maintain or upgrade system software efficiently because all the computers are network accessible. The overhead for the installation and maintenance of the Octopus software does not significantly increase the operating cost for Foo Financial, since the acquisition cost and updates are of small size and complexity.

In this scenario, there may be small initial cost in adding the Octopus software, but the 84 hours of idle computer resources are a real gain on the investment if we can use them. Therefore, we can get more return from our investment by using the available resources through Octopus compare to the current usage. Alternative methods to solve these problems such as buying more hardware and necessary software impact Foo's operating costs. For example, getting a Unix server and buying or building new software to solve the specific problem is potentially expensive.

7.1.3 Be Independent of Problem Type

Since Octopus is a framework for building P2P distributed computing applications, its architecture is not coupled to a particular type of problem set. Any large problem (or small problem for that matter) that can be divided into smaller work units can be solved using the Octopus architecture. For example, the FP problem is prototypical of one type of problem that business might need to solve. However, the Octopus architecture is not tied to the FP problem. For example, we can generate a Monte-Carlo simulation of a different problem, find an extra-terrestrial signal from huge quantities of data gathered by telescopes around world, or find a DNA sequence from data gathered by DNA sequencing machines.

Our experimental test scenario is to have two publishers to publish two different kinds of problems. Our prediction is that Octopus shall have no problem in distributing the problems and returning solution(s).

7.1.4 Be Scalable

In a distributed environment, available resources can be added at will - the notion of scale. In fact, we are seeking volunteer systems to help solve the problem. Hence, we can tackle harder, resource intensive problems with minimal concern for resource constraints as long as we have an ample supply of volunteers. If the problem is too large for the team of computers, we simply add more computers from the volunteer pool without having to redesign, rebuild and redistribute our application. In general, if there are many problems to solve, we scale more peers to the network to solve problems. Similarly, if there are fewer problems, all the idle peers solve the same problem, in which case we can take peers out of the volunteer pool scaling down the system.

Octopus is designed to scale with increasing problem load by dynamically augmenting hardware resources. The contrary is true in the case of scaling down. Here our position is that if there more problems presented, we add volunteers to process the

problem. When we add a time dimension to our claim, it raises the questions, does the Octopus architecture scale linearly, and how does the overall system power grow with the number of processors? Due to overhead in processing and the mixture of hardware, the scale probably will not be $\theta(n)$. We do not have control over what hardware to add since these resources can be considered simply a set of members from the volunteer pool. Therefore, the new volunteer can be fast or slow, and no effort is made to prioritize volunteer selection based upon speed. That is, computer system speed is multidimensional and must consider ram type, disk type, CPU, and configuration and deployment settings. Naturally, the type of hardware is related to how fast it can process a problem. Similarly, there are processing overheads that make it hard to achieve truly linear scalability.

We want to measure scalability as a graph of the number of volunteers versus problem size (with time being the third variable). For our test, we first increase the number of volunteers to solve a set of problems then observe the time to solve the problem. For our second test, we increase number of problems with a given set of volunteers to observe how long it takes to solve the problem. The three dimensional graph gives us an indication of the scalability of Octopus for a given volunteer set. It is entirely reasonable to get different results with a different mix of hardware. As Hennessy and Patterson [10] point out, to get the same result in a performance measurement, we would have to maintain the same conditions. Even slightly tuning of a piece of software can change the performance test results by a large margin.

7.1.5 Be Fault Tolerance and Availability

"Non-distributed systems typically have little tolerance for failure; if a stand-alone application fails, it terminates and remains unavailable until it is restarted" [7]. A stand-alone application may be less graceful in handling failures, as the hardware is a single point of failure. However, a distributed environment is not dependent on a single hardware or software system. It can handle the failure gracefully by removing the

hardware as a single point of failure. If one computer fails, many computers can take on the task. Hence, the fault tolerance and availability is built into the environment by design rather than by product. We discussed in section 5.3 on how our design incorporates fault tolerance.

Distributed computing is a natural system for working on our large-scale problem. A distributed environment is not only simple but also elegant in design - a set of processors work independently to solve problems.

In Octopus, we host multiple DCS's (Distributed Communication Space) in many different hosts to make Octopus available for use at any time.

Our claim is that fault tolerance and high availability are built into the environment by architecture or design. We substantiate this claim by quantitative measures. Our test scenario is to crash or remove publishers and acquirers in a systematic and repeatable way. We bring the publisher back online to observe if the publisher can still function as per the specification. Our second test scenario is to observe if there is any impact on providing solutions to problems when acquirers abruptly disappear. Our prediction is that when a publisher comes back online, it should be able to perform normally - i.e. obtaining results from DCS and processing them without incident. We shall observe no impact due to the disappearance of the acquirer, as other volunteers will pick up the problem that the acquirer needs to process.

7.2 Risk

Risk is uncertainty, which is inherent part of natural systems. Risks cannot be totally erased, but only mitigated. Since risk is part of nature, and return is inversely related to risk, there is benefit from taking a calculated risk. Risks are part of the Octopus architecture, which can be minimized without incurring major cost. We target the Octopus architecture to be used inside a protected corporate environment where many security issues are managed by the system. However, this does not mean that Octopus

is immune from risks. If Octopus is used in open network, the Octopus environment needs to be tightened to give security to peer computers.

There are risks in using distributed computing, especially P2P distributed computing, because the computers are sharing their available system resources. The very notion of P2P distributed computing is to solve a single problem by utilizing unused resources that are in networked environment. To resolve this contradiction, most implementers of P2P networking allow only certain resources of the computer to be visible to peer computers. For example, the SETI@HOME server provides all the software to process a problem. In other words, the SETI@HOME client is a customized application developed specifically for processing SETI@HOME problems.

Since Octopus is an open framework for solving problems, it is exposed to possible risks from hackers or rogue applications. Therefore, it is very important that we address these different types of risks. The following are broad categories of risks:

1. Software risks
2. Hardware risks
3. Network risks
4. Security risks

We discuss each in turn in the following sections.

7.2.1 Software Risks

We categorize the risk incurred due to software errors, software loopholes that allow any component to execute a program, and problem complexity in this subsection. We talk about risk due to use of the network in section 7.2.3. We describe each risk component of software risks as follows.

Software errors cause an application to have an unintended outcome. A common software error is that when an application gets to that instruction, in a worst-case scenario, it may even crash the application. For example, in our portfolio problem,

we compute market value from the number of holdings of an instrument multiplied by the price of that instrument. Here, if we have divided instead of multiplied, we get catastrophically different results. This causes an evaluation error. This is a simple application error that has an unintended outcome. An example of an application crash is that often in Java programs, when an instruction is expecting a value of a variable; but the variable is null, then the Java application crashes with a null pointer exception.

Software loopholes are defined as allowing an external component to execute a program. Part of the Octopus environment or framework allows executing any software in support of seeking a problem solution. A publisher can define a problem that invokes internal programs of a system to damage the peer computer environment. This is not a grave problem if the program is a pure Java program. If we allow only a pure Java program to be executed, then the Java *ClassLoader* can be used to minimize the risk. We discuss how the Java *ClassLoader* can increase security in the Security Risk section 7.2.4.

Problem complexities also bring about risk in solving a problem. A solution with one component is less complex problem than a solution with n components. As problem complexity rises, the probability of getting the problem solved decreases. This is because a complex problem requires more resources and possibly a longer time to solve. Thus, a volunteer who acquires a complex problem may not be equipped computationally to handle it. This is a risk in Octopus software.

We discussed software risk. In the following section, we discuss hardware risk.

7.2.2 Hardware Risk

We categorize the risk incurred due to hardware malfunction in this subsection. A hardware malfunction can be caused by a device failure or by overuse of a device. Device drivers are software components that are often the culprit when they are unable to respond quickly enough and can eventually take the underlying hardware offline.

We can list many events where hardware fails. A hard drive malfunction crashes a

computer. This is a hardware risk because any device in a computer can fail either due to external events such as electrical surge, or due to faulty manufacturing. If hardware is damaged by an electrical surge, then it is not available to solve problems.

Software can cause hardware malfunction by overusing the resources. For example, software might use memory to such an extent that it cause a system to crash. This is a daily occurrence in some cases. This is often called a memory leak in software. Similarly, a computer crashes if a hard drive is over-allocated and unable to write more data.

The hardware risks are inherent in P2P distributed computing. The very notion of P2P distributed computing involves sharing hardware resources. When resources are shared among peers, there is always a possibility that a peer's resources may be over-extended. To remove this risk of excessive consumption, DCC allows users to configure to allocate certain amounts hard disk space. Therefore, a peer maintains allocation information about resources for use by Octopus. In other words, the peer computer is in full control of its resources. This means that if the peer computer is not interested in sharing its resources, it can discontinue its participation or reduce its offerings to Octopus at any time.

Octopus uses CPU resource only when the peer is idle. This is done through the use of an application such as a screen saver. Thus, the user maintains full control of its CPU usage. In the Octopus architecture, peer computers themselves can easily control the hardware risks borne by software. This risk does not tend to be a critical factor to the system.

Since users are in full control of their hardware, they can limit network access to prevent overuse of the network. They can configure the network access intervals, and also limit access to the network during certain times of the day. The network is an important resource as Octopus system depends upon it heavily. Therefore, we give it a separate subsection 7.2.3 in which we discuss in detail the use and overuse of the

network.

7.2.3 Network Risk

We discuss the risk due to network use in this subsection. The network risks are due to the use of the network capacity by a peer or group of peers in the network. P2P distributed computing depends on using the bandwidth of a network when either communicating with peers or exchanging data between peers. However, the only time the network bandwidth is used is when getting the problem and data, and return of solution. Other requests, for example, checking for problems does not require much use of network bandwidth. There is an issue with network over-utilization. We discuss network usage and possible solution against over use of bandwidth in following paragraphs.

One way to reduce the risk of network overuse is to put a limit on the size of data transmissions. At the same time, we can throttle the network use to ease congestion. Current network infrastructure can support downloading 10 to 15 Mega Bytes in under 30 minutes of time. We can put a limit at 10 Mega Bytes data and reduce the problem size. If a publisher is requesting the solution of a problem containing a data set larger than the limit, then we allow acquirers to access data from a location specified by the publisher. This strategy reduces the consumption of network bandwidth; however, it does not eliminate the risk.

The network use that we described above is intended to use network bandwidth differently from a *Denial of Service* attack. Denial of Service attacks can occur when a web server receives massive requests from networked clients.

Octopus does not prevent a publisher from defining the processing such that it requires applications in the peer computer to be involved. This is a strength and a risk of Octopus. A malicious publisher can do harm to a peer computer, for example by disabling devices. If it is deployed within a private wide-area-network (WAN) or within a virtual-private-network (VPN) internal to a company, then the risk is

reduced. The network is protected, and most of the users are known users bounded by corporate laws in using the system. The risk is grave if we use Octopus in the Internet with an open network where anybody can join and leave at will. For this type of use, Octopus must be restricted to use only the Java programming language and not allow invoking other internal applications in the peer computer. This reduces the risk considerably. We can restrict by using a tailored Java Classloader so that we can load certain classes only. We discuss the Java Classloader in the Security Risk section 7.2.4. Also with the new release of Jini, the specification improved the security model that allows dynamic authentication and authorization, which we can use to secure our network. We discussed the Jini security model in Chapter 3.

So far we discussed risk relating to software and hardware. Although risks relating to the security are part of software and hardware risks, we describe security risks in a separate section 7.2.4, as security risk is an important topic. In the following section, we discuss how a malicious user can exploit the system and how to restrict such users.

7.2.4 Security Risk

A security risk is a possibility of an external user or a program breaking into a system either to compromise information or to damage the environment. The security breach occurs when a user, often known as a hacker, exploits glitches in software and hardware to break into the system thereby causing damage to the system or its information.

Since Octopus works by allowing peers to cooperate in solving a problem, a part of the problem-solving process requires executing some programs in volunteer computers (hardware). On the surface, the volunteer computers are breaching their security by allowing some foreign programs to run. How can a volunteer be sure that the program it was provided was not an intentionally malicious or error-infested program? This is the security risk in Octopus.

Since Octopus is written in Java technology, the security risk is manageable because Java technology uses a Java Virtual Machine (JVM) to run programs. Each

JVM has a Java *ClassLoader*, which is a Java class *ClassLoader*. *Classloaders* are a fundamental module of the Java language [8]. A classloader is a part of the Java virtual machine (JVM) that is responsible for finding and loading class files at run time into the JVM. The JVM allows Java programmers to write programs that are independent of operating systems (platforms). Hence, unlike programs written in C, C++, or other languages, Java programs are not a single executable file but are composed of many individual Java class files. These class files are loaded into memory on demand as needed by the application by the *ClassLoader* class.

Dynamic class loading is an important feature of the JVM. It provides the JVM with the ability to install software components at run-time. It has a number of unique characteristics. Dynamic class loading maintains the type safety of the JVM by adding link-time checks. Moreover, programmers can define their own class loaders that, for example, specify the remote location from which certain classes are loaded, or assign appropriate security attributes to them. *Classloaders* can also be used to provide separate name spaces for various software components. For example, a browser can load applets from different web pages using separate class loaders, thus maintaining a degree of isolation between those applet classes.

Classloaders have a hierarchy with parent classloaders and child classloaders. The relationship between parent and child classloaders is analogous to the object relationship of super classes and subclasses. The bootstrap classloader is the root of the Java classloader hierarchy. The Java virtual machine (JVM) creates the bootstrap classloader, which loads the Java development kit (JDK) internal classes and java.* packages included in the JVM. (For example, the bootstrap classloader loads java.lang.String.) The extension classloader is a child of the bootstrap classloader. The extensions classloader loads any JAR files placed in the extensions directory of the JDK. This is a convenient means to extending the JDK without adding entries to the classpath. However, anything in the extensions directory must be self-contained and can only refer to

classes in the extensions directory or JDK classes.

Every application receives its own *ClassLoader* hierarchy. The parent of this hierarchy is the system classpath class loader. This isolates applications so that application A cannot see the class loaders or classes of application B. In the hierarchy of class loaders, no sibling or friend concepts exist. Application code only has visibility to classes loaded by the *ClassLoader* associated with the application (or module), and classes that are loaded by *ClassLoaders* that are ancestors of the application (or module) classloader.

The *ClassLoader* does not load classes other than Java classes. This fact reduces the ability of malicious programs to run programs other than Java programs. On top of this, *ClassLoader* provides a restriction on what types of Java classes it can load. Therefore, this technology allows us reasonably to secure the member computer from outside attacks. These facts allow us to manage the security risks.

With the understanding of benefit and risks, we develop the Octopus architecture in the next Chapter. We describe in detail the implementation of the Octopus architecture using the JavaSpace technology.

8 Implementation

In this Chapter, we present a detailed discussion of the development of the Octopus architecture and the resulting system. We demonstrate through a couple of simple problems that Octopus can be used to solve problems as designed. We relate the technologies that we described in previous chapters 3 and 4 with the theory that we presented in chapter 5.

In the Development section 8.2, we give an overview of the software development process that we followed in the production of the Octopus application. The subsections present the Octopus development phases that correspond to the various stages of a software development process. In the Sequence Diagram section 8.2.1, we present the sequence diagram pertaining to the Octopus framework. In the Object Diagram section 8.2.2, we present the UML objects of Octopus system. In the Implementation Details section 8.2.3, we present and describe the Java classes that define important parts of Octopus.

In the System Configuration section 8.3, we discuss the software configurations and deployment details of the Octopus. This involves setup and configuration files, mandatory software libraries, and the Octopus installation procedures. In section System Deployment 8.4, we discuss hardware and network prerequisites and configurations needed to execute the Octopus system.

8.1 Implementation

In this section, we bring together the technologies and the design that we described in chapters 3, 4, and 5. We revisit relevant topics in technologies and theories to help describe our implementation. We begin by recapping the technology that we presented in chapters 3 and 4.

As a part of our technology presentation, we discussed Jini, RMI, and JavaSpace. Once installed in a machine, a JavaSpace is a network-accessible shared memory area,

where members of a networked community can read objects from and write objects to the JavaSpace. There are two types of JavaSpace: transient and persistent. We use a transient JavaSpace as developed and distributed by Sun Microsystems for our purposes of our demonstration. However, Octopus shall use persistent space in production environments so that all the objects can be restored in an event of failure. In the Octopus design, the Distributed Communication Space (DCS) is the JavaSpace. The DCS acts as a message hub for the members of the community.

A ticket is a standard language that the members use to communicate among themselves. A ticket is a Java object that implements the Jini **Entry** interface of the *net.jini.core.entry* package. We described in Chapter 4 that any Java objects that are written in the DCS are required to implement the **Entry** interface object. This is part of an effort to standardize objects in the DCS. The standardization of objects allows all the participating members to communicate in a single object language. In Octopus, there are two types of tickets: a problem and its solution. An acquirer acquires a problem from the WPC (Work unit Publication Channel), a CFIFO queue, while a publisher gets solution tickets from SPC (Solution Publication Channel) in the DCS. A publisher builds a problem ticket and then writes to the WPC in the DCS. An acquirer builds a solution ticket and then writes to the SPC in the DCS.

As a part of the initialization of the DCS, a transient JavaSpace from Sun Microsystems is installed in a target machine. Then the WPC is initialized in the DCS (JavaSpace) through a script. The DCS is then ready for use. Since a JavaSpace is a Jini service, the part of initialization process of the DCS (JavaSpace) is to initialize the Web services, the RMI server, and the Lookup Service. We discuss the implementation of WPC and SPC in the Implementation Details section 8.2.3.

The Distributed Client Communication (DCC) is the client application that resides in a client machine. It is the client application that

1. connects to the DCS for publication,

2. acquires problems, and

3. returns solutions.

The DCC is a single application that allows a member to be either a publisher or an acquirer. The DCC is implemented using Plain Old Java Objects (POJO). Java technologies allow us to extend existing application programming interfaces (API) to build the DCC. The user interface is developed using SWING components using the Model-View-Controller (MVC) design pattern.

We chose Java as a technology for development because of the availability of API's and software infrastructures. Moreover, Sun Microsystems is freely providing JavaSpace, a fully developed product along with API's. This allows rapid and inexpensive software development by taking advantage of open source materials. In business terms, this means faster delivery of software with lower cost.

We have described the technologies used to build the Octopus application. Now we present a problem that we use in demonstrating Octopus. For demonstrating purposes, we chose two problems: i) finding a stock quote from a given ticker symbol, and ii) a Monte Carlo simulation of a financial portfolio consisting of four instruments. The demonstration of the system with those problems allows the Octopus framework to show that it works as described in the design of the Octopus.

In the following section, we describe the process of software development in building Octopus, and the details of the technical implementation of the Octopus design.

8.2 Development

In this section, we divide the development phases into subsections in which we describe each phase of the Octopus development.

We employed a standard Rational Unified Process (RUP) [12] iterative approach for software development. We gather requirements, then design, develop, and test the application in an iterative fashion. As the word 'iterative' suggests, during the software

development, we cycle through each of those phases. For example, in the first phase, we gather requirements, and then proceed to the next phase, the design. Once we have our first draft of the design, we go back to the requirement phase and review the requirements. We revise the requirements according to the software system review and audits. Then we proceed to the design phase again. Once we have a stable design, we start the development phase. When the first phase of the development is complete, we revisit the requirement and design phases, and then return to development. The revisions are incorporated in each phase of the process. This process of iteration is done until the software is released in its production form. Once the software is released, the same process is followed for new releases.

The phases in the process map to chapters in this paper as follows. In the Chapter 5, we described the requirements in section 5.1. Then we described the Octopus design in section 5.2. In Chapter 8, we focus on the development phase, which is subdivided into three sub-phases: sequence diagrams, object diagrams, and implementation. We describe these sub-phases in subsections of this section.

In the subsection Sequence Diagrams 8.2.1, we capture the workflow and functions of major components in the system. The diagram shows how sequences of events occur at an object level - object interaction. It also describes the responsibility of an object at a higher level of abstraction and the interaction with other objects.

In the subsection Object Diagrams 8.2.2, we capture the object relationships. The diagram shows the relationships and hierarchy of objects. It is not intended to capture detailed interactions that are captured by the sequence diagram.

In the subsection Implementation Details 8.2.3, we present different code fragments to illustrate how the technology brings the Octopus architecture into life. In short, Chapter 5 is about requirements and design while Chapter 8 is about implementation of the design in Chapter 5. The sequence and object diagrams are used as a guide for development, which we discuss in implementation details section 8.2.3.

In the following subsections, we delve deeper into the code base of the Octopus. The notation that we follow is: Java classes and objects are **Object** without parenthesis, and method names are *method()* with parenthesis (we omit any parameters to improve readability). Variables are simple `variable` without parenthesis.

8.2.1 Sequence Diagrams

In this section, we present sequence diagrams. These sequence diagrams are driven from use cases that we presented in Chapter 5.

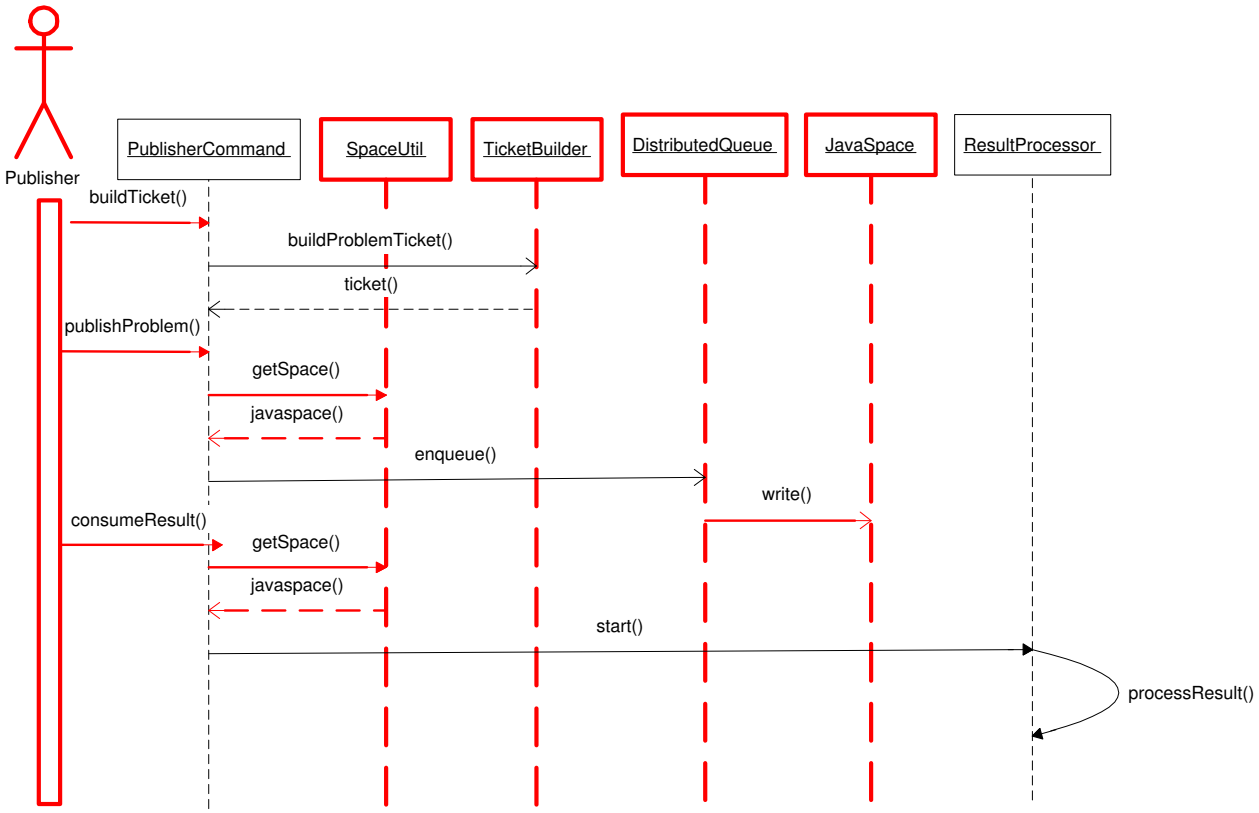

Figure 15: Sequence diagram of publisher

Figure 15 shows the sequence diagram of a Publisher. In the diagram, a main program or a user interface makes a call to **PublisherCommand** to publish a problem. First, the **PublisherCommand** object makes a call to the **TicketBuilder** to build a ticket with a problem. Then, it asks the **SpaceUtil** to find a DCS. Finally, it asks

DistributedQueue to queue the problem ticket.

When consuming results, a main program or a user interface makes a call on the **PublisherCommand** to the *consumeResult()* method. The **PublisherCommand** then asks the **SpaceUtil** to find a DCS. It asks the **ResultProcessor** to process results, which involves looking searching **ResultEntry** in the DCS. If it finds a **ResultEntry**, it processes the results set encapsulated in the **ResultEntry**. The obtained results are then displayed in the publisher view.

In Figure 16, we present the sequence diagram for an Acquirer. In the diagram, the **ConsumerCommand** asks the **SpaceUtil** to get the DCS. Then, it asks the **DistributedQueue** to get a ticket from the DCS. Once it gets a ticket, it asks the **ProblemProcessor** to process the ticket and generate a result. It receives an object of **ResultEntry** from the **ProblemProcessor**, which is then published to the DCS.

We have shown the sequence diagrams. Now we move to the Object Diagrams to describe the organization of objects in the Octopus application.

8.2.2 Object Diagrams

In this section, we show package layout and object diagrams of Octopus.

Figure 17 shows the package layout of Octopus. The package *com.shirishranjit.octopus* is the root. The *common* package consists of classes common to Octopus. Classes in this package and sub packages make library classes for Octopus. The *util* package contains utility classes, which is also part of library classes for Octopus. The *domain.ticket* package contains classes regarding problem and solution tickets. The packages *dcc* and *dcs* are the users of domain packages.

Figure 18 shows the package layout for DCC. The *dcc* package contains *acquirer*, *gui*, *publisher*, and *util* packages. The package *gui* contains GUI classes, while *util* contains utility classes only for the *dcc* package. The package *acquirer* contains classes that make an acquirer. The package *publisher* contains classes that create a publisher.

Figure 19 shows the packages that make DCS. The *gui* package contains classes for

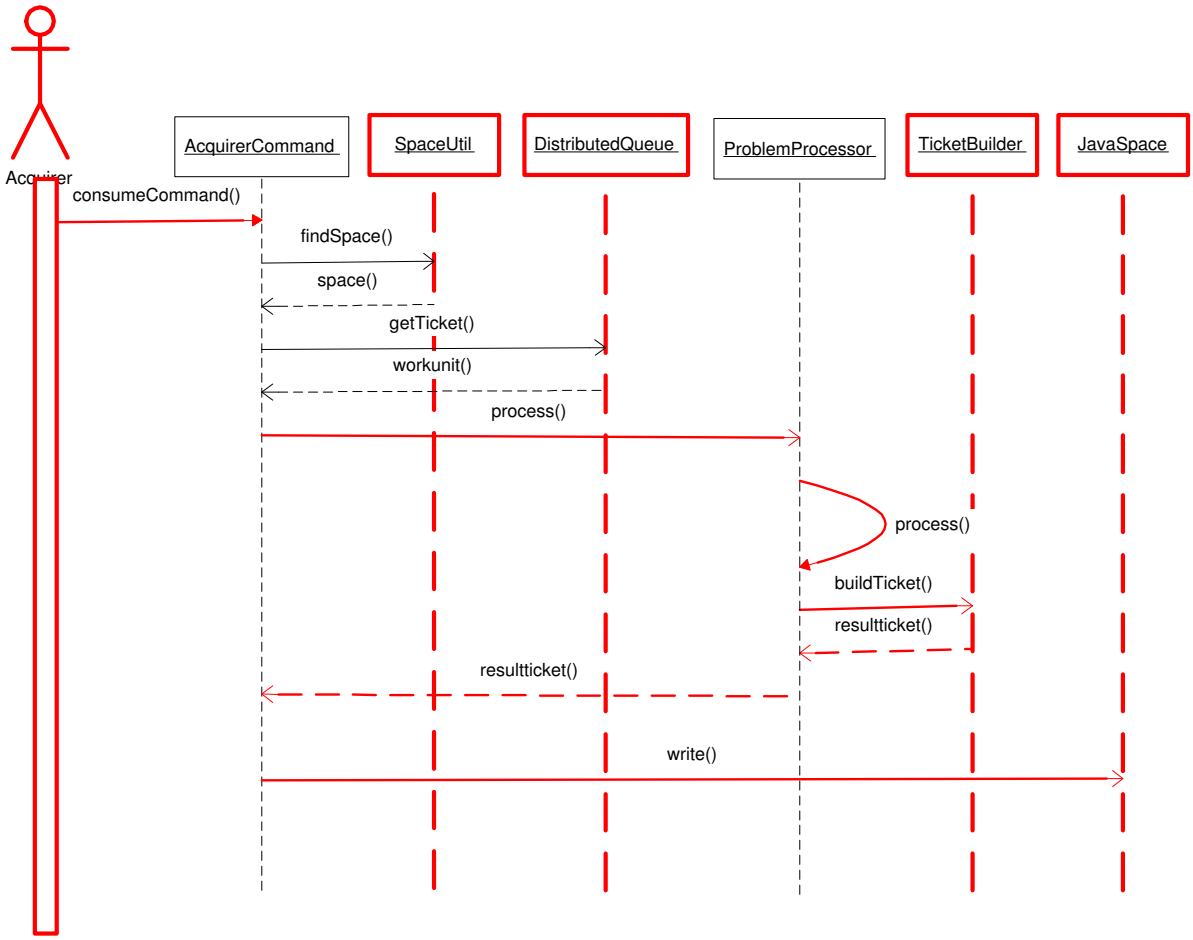

Figure 16: Sequence diagram of acquirer

the monitor GUI. The *queue* package contains classes for creating a distributed queue.

Figure 20 shows objects hierarchy for distributed queue. **Position** class is the parent to classes **Head**, **Tail**, and **QueueElementEntry** classes. **Head** class represents the head of the queue, while **Tail** class represents the tail of the queue. **QueueElementEntry** class represents the elements in the queue.

Figures 21, 22, 23, 24, and 25 shows the object relations and hierarchies.

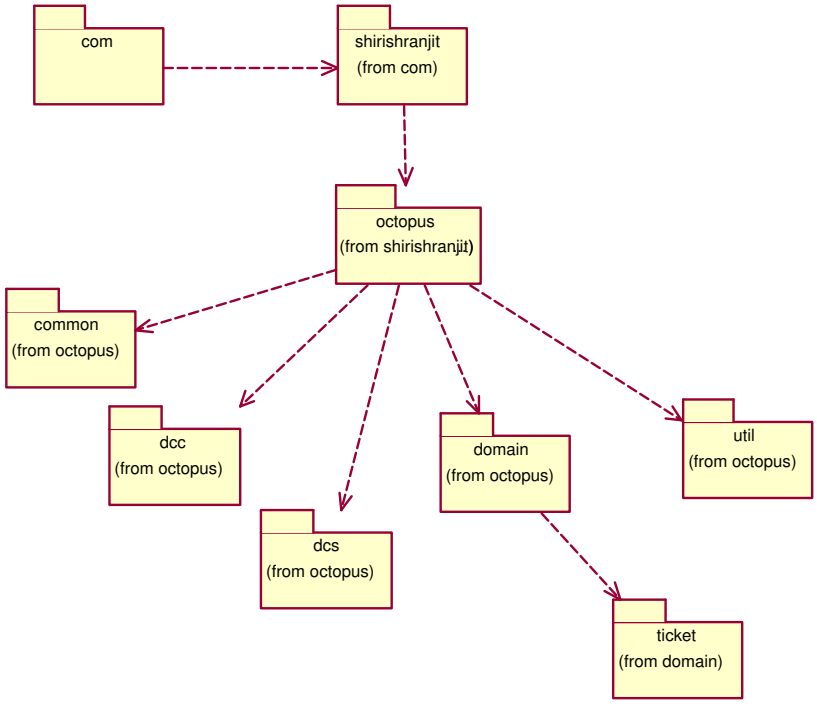

Figure 17: Overall package layout

Figure 21 shows the DCS objects relationship and their hierarchy.

Figure 22 shows the DCS and Java objects relationship and their hierarchy.

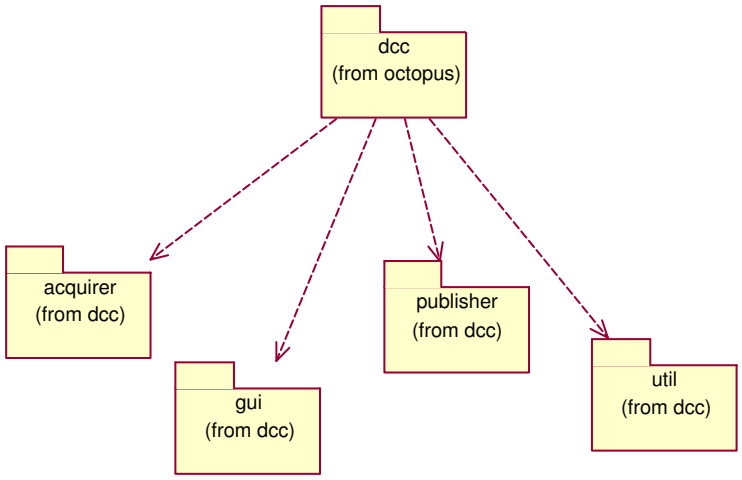

Figure 18: DCC package layout

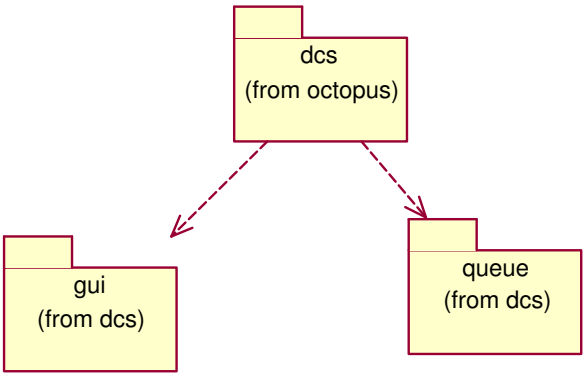

Figure 19: DCS package layout

Figure 23 shows objects relating to result entry and their relationship and hierarchy.

Figure 24 shows the DCC publisher objects relationship and their hierarchy.

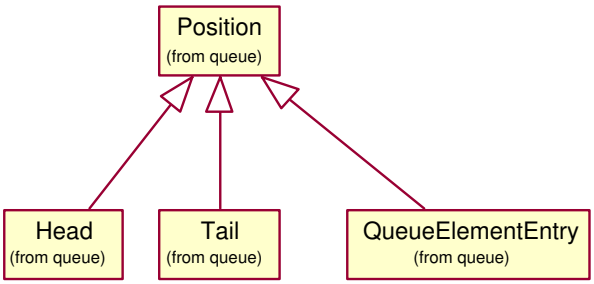

Figure 20: Queue objects hierarchy

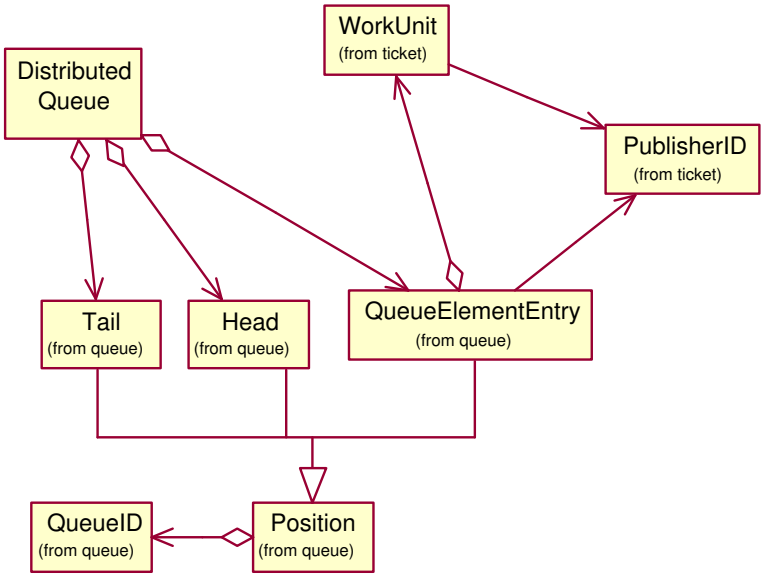

Figure 21: DCS objects hierarchy

Figure 25 shows the DCC acquirer objects relationship and their hierarchy.

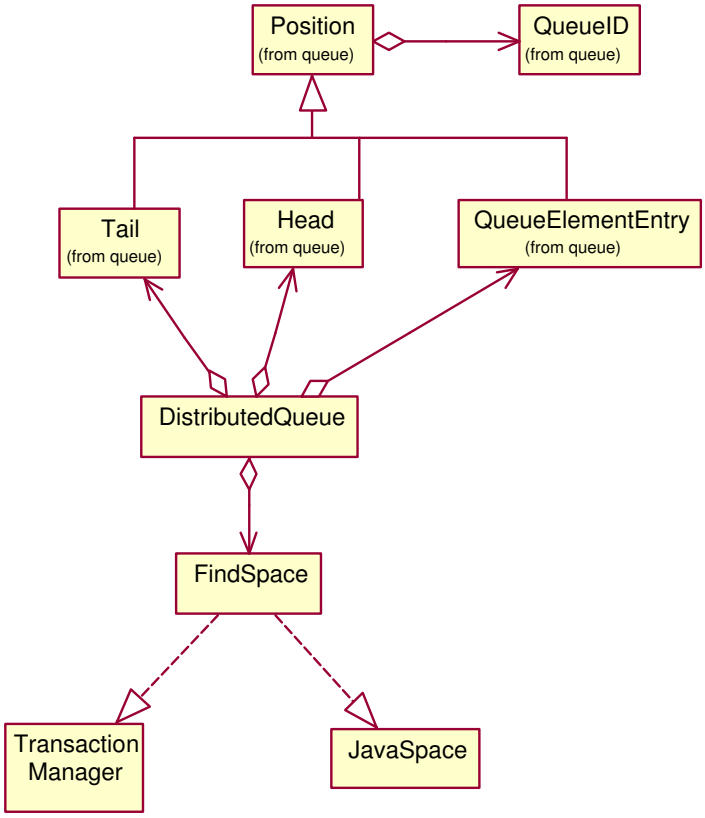

Figure 22: DCS and JavaSpace objects hierarchy

For detailed object diagrams and API's, please check the javadoc in the attached media.

In the following section, we describe implementation details of Octopus.

8.2.3 Implementation Details

In this section, we describe most relevant objects and methods that make up Octopus. Before we consider code fragments, we explain the implementation of the WPC and SPC queues in a DCS. Then we explain code fragments for finding a DCS in the **Find Space** section, as finding a DCS is central to the Octopus architecture. Then, in **Build Ticket** section, we describe how a producer/acquirer builds a ticket to write to a DCS. These two are the fundamental functions in the Octopus. Then we describe a publisher code in **Publisher Command** and an acquirer code in **Acquirer Command** sections.

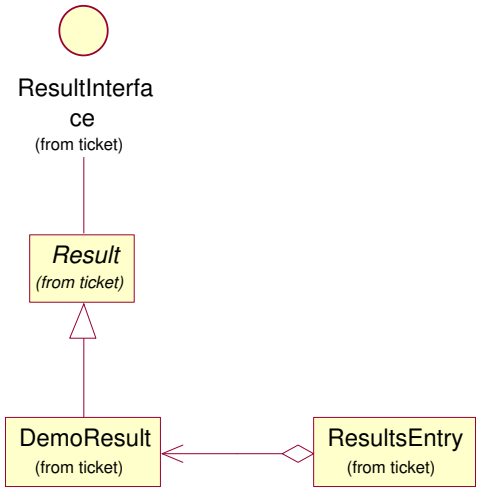

Figure 23: Result objects hierarchy

A DCS is a JavaSpace. Therefore, WPC and SPC queues reside in the JavaSpace. The WPC is a CFIFO queue implemented as a distributed queue in the JavaSpace. We use the Jini **Entry** interface from the package *net.jini.core.entry* in creating our distributed queue.

SPC is a queue-like structure. We implement the SPC as a **ResultsEntry** object that implements the Jini **Entry** interface to take advantage of the JavaSpace specification. Therefore, once a **ResultsEntry** object is written to the JavaSpace, it can be queried to find that object given the **ResultsEntry** objects member variables. An acquirer publishes a **ResultsEntry** with the publisher id that it received in a work unit. This allows the publisher to find results pertaining to it by using the query methods of JavaSpace.

DCS is implemented as a JavaSpace that contains a distributed queue. The distributed queue is initialized such that the head and tail of the queue counter are set to 0 indicating there is no element in the queue. The head and tail of the queue implement the *net.jini.core.entry.Entry* interface. An Element represented as a **QueueElementEntry** in the queue have also a position counter that indicates its position in the queue.

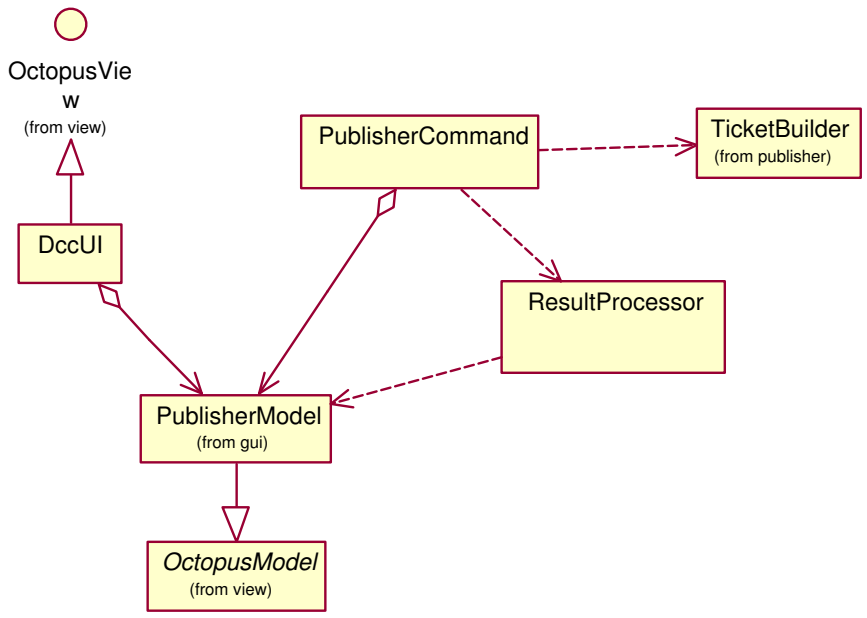

Figure 24: DCC Publisher objects hierarchy

The **QueueElementEntry** also implements the *net.jini.core.entry.Entry* interface. A work unit is wrapped in the **QueueElementEntry**. When an acquirer takes out an element from the JavaSpace, the element is taken off the head of the queue and added back at the tail of the queue. This operation accomplished by incrementing both head counter and tail counter, and updating the position counter in the element.

The implementation of DCS is a JavaSpace. Therefore, please refer to the JavaSpace javadoc for the code explanation published by SUN at *http://java.sun.com*.

Now we present the DCC code fragments and their descriptions. Before going into details of code fragments, we point out that we follow Java Coding Standard published by Sun Microsystems with some modification to the standard.

Find Space: Finds a DCS - The **FindSpace** is a utility class that provides methods to find a DCS, the primary task in publishers and acquirers. For example, an acquirer must find a DCS before requesting a problem and processing it.

The following method *getSpace()* in the **FindSpace** returns a **JavaSpace** if the

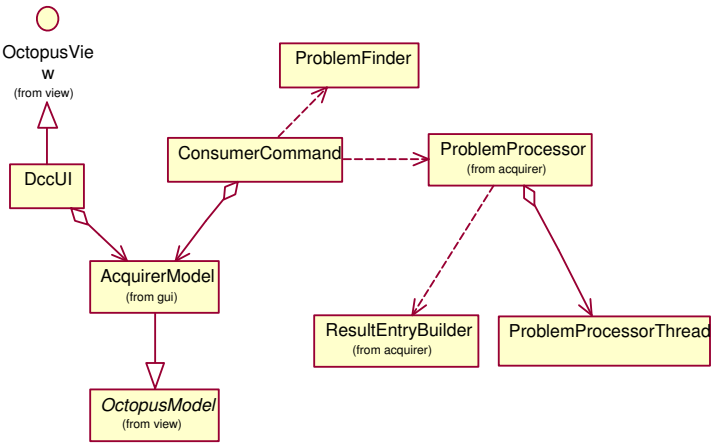

Figure 25: DCC Acquirer objects hierarchy

FindSpace object has `space`, a **JavaSpace**, cached in it. Otherwise the method makes a call to *waitForSpace()* to look up the specified space.

```
/**
 * Return the space
 * means it hasn't yet appeared. Call <code>waitForSpace()</code> at
 * least once before using this method.
 * @return JavaSpace returns a handle to a space.
 */
public JavaSpace getSpace() {
    if (space == null) {
        space = this.waitForSpace();
    }

    return space;
}
```

The method *waitForSpace()* returns **JavaSpace** that it finds in its network. The method calls *lookupServiceByType()* for finding `serviceTypes`, a list of class names, in the network. In a Lookup service, a JavaSpace is stored as a **ServiceTemplate**. Therefore, we look up a `jsp`, a **JavaSpace** as a **ServiceTemplate**. Once we receive a `item`, a **ServiceItem**, the `item` is type cast into a **JavaSpace** class. The way it is written, the method returns only one **JavaSpace** although it may find many.

```
/** Finds a JavaSpace in the network. It makes a request to method
```

```
     * lookupServiceByType with given serviceTypes as a JavaSpace.
     * @return A reference to the JavaSpace service located.
     */
    private JavaSpace waitForSpace() {
        Class serviceTypes[] = { JavaSpace.class };
        JavaSpace jsp = null;
        try {
            ServiceItem item = lookupServiceByType(serviceTypes, null);
            jsp = (JavaSpace)item.service;
        } catch (RemoteException re) {
            LOGGER.log(Level.SEVERE," Remote exception space ", re);
        } catch (IOException ioe) {
            LOGGER.log(Level.SEVERE," Exception in space "
                +" discovery: ", ioe);
        } catch (Exception e) {
            LOGGER.log(Level.SEVERE," Exception space ", e);
        }
        return jsp;
    }
```

The following method *lookupServiceByType()* finds an `item`, a **ServiceItem**, from a Lookup service in the network. It looks up a **JavaSpace** as a `tmpl`, a **ServiceTemplate**, that contains the JavaSpace class name. The method first creates a `sdm`, a **ServiceDiscoveryManager**, with a **LookupDiscovery**. Then using the `sdm` instance, it looks up the given `tmpl`. The `sdm` returns an `item`, a **ServiceItem**. The method will try for `NUM_ITERATION` times to find the given **tmpl** before quitting.

```
    /** This method uses the ServiceDiscoveryManager to locate the
     * particular type of service specified by the caller. The method
     * either finds the item or goes through specified number of
     * iterations before returning a null item.
     *
     * After each loop in while, it sleeps for specified time period.
     *
     * @param serviceTypes Array of classes which our desired service
     *                     should implement. If a service matching one
     *                     of these type is found, it is further
     *                     scrutinized by the ServiceItemFilter.
     * @param sif Either null or an object that implements
     *                     a method which returns a boolean to indicate
     *                     whether this service item is acceptable.
     * @return A service item matching our class criteria and also
```

```
     *         acceptable to any ServiceItemFilter passed.
     * @throws IOException throws an IOException
     */
    private ServiceItem lookupServiceByType(Class serviceTypes[],
    ServiceItemFilter sif)
    throws IOException {
        ServiceDiscoveryManager sdm = null;
        ServiceTemplate tmpl =
            new ServiceTemplate(null, serviceTypes, null);
        ServiceItem item = null;
        String groups[] = new String[1];

        groups[0] = new String(getLookupGroup());
        sdm = new ServiceDiscoveryManager(
                    new LookupDiscovery(groups), null);
        int i = 0;
        while ((item == null) && (i < NUM_ITERATION)) {
            try {
                item = sdm.lookup(tmpl, sif, 10000);
            } catch (InterruptedException intr) {
                LOGGER.log(Level.SEVERE, " Interrupted...");
                continue;
            }

            if (item == null){
                try {
                    LOGGER.log(Level.INFO, " item not found yet. "
                        +" Interation: "+i);
                    Thread.sleep(WAIT_TIME);
                }
                catch(InterruptedException ie) {
                    LOGGER.log(Level.SEVERE, " The FindSpace is "
                        +" interrupted.", ie);
                    continue;
                }
            }
            i++;
        }
        return item;
    }
```

Once we have a handle to the JavaSpace, we need a **TransactionManager** for doing operations in the JavaSpace (DCS). The following method *waitForTransMan()* allows us to get a handle for a **TransactionManager**. Just like a **JavaSpace**, a **TransactionManager** is also stored in a Lookup service. This

method also calls the *lookupServiceByType()* with the `serviceTypes`, a list of **Class** objects, of **TransactionManager** objects.

```
/**
 * Finds a transaction manager in the network. If none can
 * be found, it returns a null.
 * @return A reference to the JavaSpaces service located.
 */
private TransactionManager waitForTransMan() {
    Class serviceTypes[] = { TransactionManager.class };
    TransactionManager tm = null;
    try {
        ServiceItem item = lookupServiceByType(serviceTypes, null);
        tm = (TransactionManager)item.service;
    } catch (RemoteException re) {
        LOGGER.log(Level.SEVERE, " Remote exception "
            + " transacation manager ", re);
    } catch (IOException ioe) {
        LOGGER.log(Level.SEVERE, " IOException in "
            + " transacation manager discovery: ", ioe);
    } catch (Exception e) {
        LOGGER.log(Level.SEVERE, " Exception "
            + " transacation manager ", e);
    }
    return tm;
}
```

We need to build tickets before we can write to the DCS. We present the following code for building a ticket.

Bulding Ticket: Builds a ticket - There are two types of tickets - problem and solution. A publisher creates a ticket with a work unit. An acquirer creates a result ticket containing results or solution of the solved problem. Both the publisher and the acquirer publish their respective tickets to DCS.

The following code block is the *buildWorkUnit()* method in the **TicketBuilder**. The method builds a work unit for a publisher.

```
/**
 * Builds a work unit ...
 * @return WorkUnit... a unit of work.
 */
public WorkUnit buildWorkUnit(PublisherID pubId){
```

```
        WorkUnit wUnit = new WorkUnit(pubId);

        return wUnit;
    }
```

It creates an instance of the **WorkUnit**, `wUnit`, that contains publisher id. Then, necessary parameters are set and returned in the object to the caller. A publisher has a choice of publishing either a stock quote problem or a Monte Carlo simulation of the value of a portfolio, or both problems. This creates a work unit with chosen problem or problems. The work unit is queued in the DCS. When the publisher chooses a Monte Carlo simulation problem, it marks the boolean variable in **WorkUnit** as true. The **WorkUnit** object has a predefined portfolio simulation model.

The following code represents fragments from **ResultEntryBuilder** that helps to build a result ticket.

```
    /**
     * The method builds ResultsEntry from the parameters supplied.
     * The ArrayList it received contains results in a string that is
     * then added to the DemoResult object.
     * @param pubId publisher id
     * @param results list of results
     * @param simResults list of simulation results
     * @return ResultsEntry object
     */
    public ResultsEntry buildResultEntry(PublisherID pubId,
    ArrayList results, ArrayList simResults) {

        if ((results.size() == 0) && (simResults.size() == 0)) {
            return null;
        }
        ArrayList resultHolders = new ArrayList();

        for (int i = 0; i < results.size(); i++) {
            String sol = (String)results.get(i);
            //a holder of the result.
            DemoResult demo = new DemoResult();
            demo.setSolution(sol);
            resultHolders.add(demo);
        }
```

```
        //add the simulation results.
        DemoResult simDemo = new DemoResult();
        simDemo.setSimResult(simResults);
        resultHolders.add(simDemo);

        return buildEntry(pubId, resultHolders);
    }

    /**
     * Builds a ResultsEntry out of a collection of
     * DemoResults containing the
     * results from processing of the given problem.
     * @param pubId publisher id
     * @param resultHolder a Collection object.
     * @return ResultsEntry object
     */
    private ResultsEntry buildEntry(PublisherID pubId,
    Collection resultHolder) {
        ResultsEntry resEnt = new ResultsEntry(pubId);
        resEnt.addList(resultHolder);
        return resEnt;
    }
```

In the **ResultEntryBuilder**, the private method *buildEntry()* takes a `pubId`, a **PublisherID**, a `results`, an **ArrayList** of results from Stock Quote, and a `simResults`, an **ArrayList** of simulation results to build the **ResultsEntry** object. The method returns a **ResultsEntry** object that contains solutions and the publisher id. The **ResultsEntry** has **PublisherID** so that publishers can search the DCS with their **PublisherID**. The **PublisherID** contains a Java **Long** object.

An acquirer command, **ConsumerCommand**, calls the method *buildResultEntry()* for building a **ResultsEntry** object. It takes a `pubId`, a **PublisherID**, a `resultHolder`, a **Collection**, a list of results containing a list of results from the simulation and the Stock Quote as a **String** data type. Inside the 'for loop', the *buildResultEntry()* creates a a `demo`, a **DemoResult** that encapsulates the Stock Quote results and `simDemo`, a **DemoResult** that encapsulates the simulation results. The method calls the private method *buildEntry()*

to construct the **ResultsEntry**.

The **ResultsEntry** object contains solution elements for both problems. The *setSolution()* method sets the solution from stock quote while *setSimResult()* sets the solution from the simulation. A publisher checks for those two elements of **ResultsEntry** object to process solutions.

We have described how to find a DCS and how to build tickets. Now let us look at how a publisher works. The following **PublisherCommand** and **Consumer-Command** in implement the *Command* design pattern.

Publisher Command: A publisher - Functions of a publisher are represented in the following **PublisherCommand**. The **PublisherCommand** has three essential methods, *publishProblem()*, *consumeResult()*, and *removeProblem()*.

The method *publishProblem()* takes a `wu`, a **WorkUnit**, object as an input parameter. It then finds a JavaSpace, and instantiates the **DistributedQueue** as a `da` to queue the `wu`, work unit, in the JavaSpace. The **DistributedQueue** object has the expertise to interact with the JavaSpace to queue the **WorkUnit**.

```
/**
 * The method builds a work unit, and then pubishes it to
 * the distributed queue (DCS). It gets a JavaSpace from FindSpace,
 * then makes a request to a DistributedQueue object to enqueue
 * the work unit.
 *
 * @param wu a work unit for publication.
 */
public void publishProblem(WorkUnit wu) {
    try {
        //get a problem

        //find a JavaSpace
        FindSpace fspace = new FindSpace();
        JavaSpace space = fspace.getSpace();
        //publish the problem in the distributed queue
        //in the JavaSpace.
        LOGGER.log(Level.INFO, " Publishing the ticket now ");
        DistributedQueue dq = new DistributedQueue(Lease.FOREVER);
        //publish the ticket here.
        dq.enqueue(wu, wu.getPublisherId());
```

```
        }
        catch (Exception e) {
            e.printStackTrace();
            LOGGER.log(Level.SEVERE, " Exception in publish "
             + " problem ", e);
        }
    }
```

The method *consumeResult()* consumes results intended for the publisher. The method first finds a `fspace`, the **JavaSpace**. Then it creates a **ResultProcessor** object as a `rp`, with the `fspace`, the `pubId`, and a `pubModel`, the **PublisherModel** objects. The `rp` gets all results from the DCS, processes, and updates the `pubModel` for viewing. Therefore, when the processing is complete, the view gets updated with the result information stored in the **PublisherModel** object. Once the publisher receives the necessary results, it removes the problem from the JavaSpace. The *removeProblem()* method accomplishes the work of removing problems published by the publisher. The method *removeProblem()* first gets a handle to a `fspace`, the **JavaSpace**. Then it requests the **DistributedQueue**, `dq`, object to remove a problem that contains the publisher's id.

The following function is *removeProblem()*.

```
    /**
     * This is a driver method in consuming results from the JavaSpace.
     * The method finds a JavaSpace, and then searches the JavaSpace
     * with its id to retrive all the results available in the Space.
     * As results are processed, the output is stored in PublisherModel
     * for View. Once the processing is complete, the View is
     * updated with the data in PublisherModel.
     *
     * @param pubModel A model for capturing the information on
     * processing of results.
     */
    public void consumeResult(PublisherModel pubModel) {

        try {
            //find a JavaSpace
            FindSpace fspace = new FindSpace();
            JavaSpace space = fspace.getSpace();
            //Wait for the result... spawn a thread that waits and
```

```
            //checks the space.
            LOGGER.log(Level.INFO, " Processing results ");
            ResultProcessor rp =
                new ResultProcessor(space, pubId, pubModel);
            rp.process();
            //Thread rpT = new Thread(rp, pubId.toString());
            //rpT.start();
        }
        catch (Exception ex) {
            ex.printStackTrace();
            LOGGER.log(Level.SEVERE, " Exception in consume result ",
             ex);
        }

    }

    /**
     * The method removes a problem that the publisher published.
     * It searches the distributed CFIFO queue with its id to remove a
     * problem at a time.
     */
    public void removeProblem() {

        try {
            //find a JavaSpace
            FindSpace fspace = new FindSpace();
            JavaSpace space = fspace.getSpace();
            //Wait for the result... spawn a thread that waits and
            //checks the space.
            LOGGER.log(Level.INFO, " Removing Problem...");
            DistributedQueue dq = new DistributedQueue(Lease.FOREVER);
            //Remove Problem with given entry
            dq.removeQueueElement(pubId);
            LOGGER.log(Level.INFO, " Removing Problem complete...");
        }
        catch (Exception ex) {
            ex.printStackTrace();
            LOGGER.log(Level.SEVERE, " Exception in remove problem",
             ex);
        }

    }
```

As we have discussed how a publisher works, the following code block shows how an acquirer works as a part of DCC.

Acquirer Command: An acquirer - The following code fragments represent an ac-

quirer function. The class **ConsumerCommand** shows an acquirer getting a problem from the DCS, then processing the problem, and writing the results back to the DCS.

This demonstration application is a simple one to demonstrate that the Octopus architecture works. Thus, the application is not sophisticated to handle termination of processing in situations such as machine being turned off in the middle of the processing. Our vision is that the application should able to recover from its last known state and start processing. This is a nice feature. However, the Octopus architecture is designed to allow such failure. Therefore, the failed acquirer can just wake up and ask for a new problem to solve instead of trying to recover the last known state and process forward.

```
/** This is the driver method of an acquirer. It finds a
 * JavaSpace, gets a problem from the JavaSpace, process it, and
 * then publishes a result into the JavaSpace.
 * @param model a model for storing information for presenting in
 * acquirer view.
 * @throws OctopusException OctopusException
 */
public void consumeCommand(AcquirerModel model)
    throws OctopusException {
    try {
        //a. first find the JavaSpace.
        FindSpace fspace = new FindSpace();
        JavaSpace space = fspace.getSpace();
        //b. get a problem from the JavaSpace
        WorkUnit wu = getProblemFromSpace(model);

        if (wu != null) {

            model.setResults(wu.toString());

            LOGGER.log(Level.INFO,
            " ConsumeCommand: what we have in wu: \n" +
            wu.toString());
            //instantiate the processor.
            ProblemProcessor pp = new ProblemProcessor();
            //c. process the problem.
            ResultsEntry re = pp.process(wu);
            LOGGER.log(Level.INFO,
```

```
                " ConsumeCommand: This is the ResultsEntry:  " + re);
                //d. write results to the JavaSpace.
                if (re != null) {
                    space.write(re, null, Lease.DURATION);
                }
            }
            else {
                model.setResults(" There is no problem to solve: "
                    + " Lets try again later.");
            }

        }
        catch (Exception e) {
            if (e instanceof OctopusException){
                throw (OctopusException) e;
            }
            else{
                e.printStackTrace();
                throw new OctopusException(" There is an exception in "
                    + " ConsumerCommand ", e);
            }
        }
    }
```

The *consumeCommand()* method first get a handle to a `fspace`, the **JavaSpace**. Then, it makes a request to the private method *getProblemFromSpace()* to acquire a work unit, `wu`. The method asks **ProblemProcessor** to process the work unit. The *process()* method processes the `wu`, a work unit, and returns a **ResultsEntry** object for publishing into the `fspace`, the **JavaSpace**. The result is written to the DCS.

The following function is *getProblemFromSpace()*.

```
    /** This method gets a work unit from the JavaSpace. The
     * method tries to get a work unit for fixed number of times. After
     * completion of each iteration, it sleeps for a fixed time
     * period. This allows the method to wait until a work unit appears.
     * If it cannot find a work unit within that time period, it
     * exits with a null work unit.
     *
     * @param model An acquirer model for storing data for viewing
     * @return WorkUnit A unit of work received from the JavaSpace
     * @throws OctopusException OctopusException All the exceptions are
     * wrapped in this exception
```

```
 */
private WorkUnit getProblemFromSpace(AcquirerModel model)
    throws OctopusException {
    try {
        boolean gotProblem = true;
        LOGGER.log(Level.INFO, " Getting the ticket now ");
        DistributedQueue dq = new DistributedQueue(Lease.DURATION);

        WorkUnit wu = (WorkUnit) dq.getTicket();

        if (wu == null){
            gotProblem = false;
        }
        //lets wait till we get a problem.
        int i = 0;
        while ((!gotProblem) && (i < 10)) {
            model.setResults(" No work unit to process... \n"
                + " Looking for new work unit....\n");
            LOGGER.log(Level.INFO, "Checking the work to do.");
            wu = (WorkUnit) dq.getTicket();
            if (wu != null) {
                LOGGER.log(Level.INFO, " WE GOT work to do.");
                model.setResults(" WE GOT work to do... \n");
                gotProblem = true;
            }
            else {
                LOGGER.log(Level.INFO, " Nothing in the queue, "
                    + " sleeping.. This is iteration: " + i);
                model.setResults(" No work unit to process. "
                    + " Iteration: " + i + "\n" + " Waiting for "
                    + SLEEP_TIME + " millisec....\n");
                Thread.sleep(SLEEP_TIME + (10 * (i + 1)));
                i++;
            }
        }

        LOGGER.log(Level.INFO, " Got something to work. " +wu
            + " number of iteration: " + i);

        return wu;
    }
    catch (RemoteException ex) {
        throw new OctopusException(" RemoteException in "
            + " getting problem from space. ", ex);

    }
    catch (InterruptedException ex) {
```

```
            throw new OctopusException(" InterruptedException "
                + " in getting problem from space. ", ex);

        }
        catch (UnusableEntryException ex) {
            throw new OctopusException(" UnusableEntryException "
                + " in getting problem from space " , ex);
        }
        catch (LeaseDeniedException ex) {
            throw new OctopusException(" LeaseDeniedException ",
            ex);
        }
        catch (TransactionException ex) {
            throw new OctopusException(" TransactionException in "
                + " getting problem from space.", ex);
        }
        catch (IllegalArgumentException ex) {
            throw new OctopusException(" IllegalArgumentException "
                + " in getting problem from space. ", ex);
        }

    }
```

The method *getProblemFromSpace()* gets a `wu`, a work unit object, from the DCS by asking the `dq`, the **DistributedQueue** object to get a **WorkUnit** from the **JavaSpace**. It will try for a fixed number of attempts. During each attempt, the method sleeps for fixed period of time. This allows the method to have an opportunity to get a work unit from the DCS. The delay time increases with each attempt by 10 milliseconds times the iteration.

The following method *process()* in object **ProblemProcessor** processes a work unit that it receives from DCS. The method instantiates an object **ProblemProcessorThread** as a `procT`, for processing the work unit. Once the processing is complete, it gets results and builds **ResultsEntry** that is returned to the caller. We have described the building of **ResultsEntry** in **Build ticket** section above.

```
    /**
     * The method takes a work unit as a parameter. It creates a
     * ProblemProcessorThread object for processing the work unit.
```

```
     * The processing is complete, the results are set and
     * ResultsEntry is generated and returned to the command object.
     *
     * @param wu work unit
     * @return ResultsEntry an object with results from processing
     *
     */
    public ResultsEntry process(WorkUnit wu) {
        ProblemProcessorThread procT = new
         ProblemProcessorThread(wu.getUserName());
        //lets set the work unit and model
        procT.setWu(wu);

        //start the processing
        procT.process();
        //lets get the results.
        String results = procT.getResult();
        ArrayList listRes = new ArrayList();
        listRes.add(results);
        ArrayList simResult = procT.getSimResult();

        //lets build the result entry.
        ResultEntryBuilder reb = new ResultEntryBuilder();
        //lets give the result entry back
        return reb.buildResultEntry(wu.getPublisherId(), listRes,
         simResult);

    }
```

The following two methods *process()* and *simulator()* in the object **ProblemProcessorThread()** are for processing the work unit. Although the object **ProblemProcessorThread()** is not implemented as a thread yet, it is designed to work as a thread. The method *process()* calls *process()* method in **WorkUnit** object. The *process()* method in **WorkUnit** finds a stock quote first and then checks if it needs to do a simulation. If it is set to do the simulation, then it asks the *simulator()* method to do the simulation.

The method *simulator()* evaluates the model that is provided in the work unit object. The method *simulator()* executes the simulation function in the `wu` object with the iteration number in `wu`.

```
    /**
```

```
 * This method processes the work unit. First it
 * gets the stock quote. Then it processes simulation.
 *
 */
public void process(){
        LOGGER.log(Level.INFO,"Processing the WorkUnit");

        wu.process();
        result = wu.getResult();
        simResult = wu.getSimResult();

        LOGGER.log(Level.INFO,"complete processing");
}

/**
 *  Simulator method. This method simulates the portfolio.
 *
 */
private void simulator(){
    Logger LOGGER = OctopusLogger.getHandle();
    LOGGER.log(Level.INFO, " Simulator processing "+
                    " the WorkUnit Simulator ");
    int simNum = wu.getNumIteration().intValue();

    simResult.add(wu.evaluateModel());

    //lets simulate the model.
    for(int i = 0; i <= simNum; i ++){
        simResult.add(wu.simulateModel());
    }

    LOGGER.log(Level.INFO, " Simulation Complete ");
}
```

We have described implementation of DCC. In the following sections, we describe system configurations and deployment of Octopus.

8.3 System Configuration

In this section, we describe system configurations of Octopus for deployment. Since Octopus is implemented in Java, there is no dependency on hardware as long as we can run a Java Virtual Machine (JVM). Therefore, the requirements are JDK 1.4 or higher and Jini Technology Starter Kit (JTSK) 2.0 or higher.

Download J2SE version 1.4 or higher from http://java.sun.com/j2se/index.jsp, and Jini Technology Start Kit version 2.0 or higher from http://wwws.sun.com/software/communitysource/jini/download.html. Install both Jini and J2SE as per their instructions. Make sure that the file *jsk-policy.jar* needs to be installed in J2SE installation as per the Jini installation instructions. The installation instructions of JTSK 2.0 are to copy the file jini2_0_001\lib\jsk-policy.jar to j2sdk1.4\jre\lib\ext. This is absolutely necessary for Jini services including JavaSpace. For DCC, only J2SE needs to be installed for its Java Virtual Machine (JVM).

The paths of J2SE, JSKT, and libraries need to be modified in scripts before they are ready for the use. We provide fully functional demonstration software in the media as an appendix to the paper. The media contains the Jini, J2SE, and fully functional scripts to run the demonstration.

Before we can use Jini services, we need to configure the Jini Technology Starter Kit. Those Jini services are as follows.

- Fiddler - a Jini lookup discovery service
- Mahalo - a Jini transaction manager service
- Mercury - a Jini event mailbox service
- Norm - a Jini lease renewal service
- Outrigger - a JavaSpaces(TM) service
- Phoenix - a configurable RMI activation system daemon
- Reggie - a Jini lookup service

In our demonstration program, we use all the services except Mercury and Norm.

We provide a fully configured, self-contained installation of Octopus in an appendix as an electronic media. This includes DCS and DCC.

In the following section, we discuss the deployment process of the Octopus system that we have configured in this section.

8.4 System Deployment

In this section, we present initialization and deployment process for Octopus - DCC, and DCS.

The Octopus software is bundled such a way that it needs to be copied to a location and a few scripts for deployment and installation. The software is self-contained and no external installation is required. All necessary tools such as JDK and JTSK, are bundled in the software. But the software is bundled only with WINDOWS compatible JDK and JTSK only.

Following are the steps necessary for deploying the whole Octopus system.

1. Deploy DCS, that is Jini, in a machine by unzipping *DCS.zip* file in a location. `jini` is the root directory.

2. Run the script `run_all_servies.bat` in the folder `jini\bin`. This batch program starts five batch programs in the bin directory that starts Jini services. The batch programs are executed in sequence in order to bring up each service. The order and the services are

   (a) launch RMI server - Phoenix

   (b) launch HTTP server to download the code base - Codebase server

   (c) launch Lookup service - Reggie

   (d) launch Transaction manager - Mahalo

   (e) launch JavaSpace - Outrigger

   Both Codebase Server (HTTP server) and Phoenix (RMI) services are running on the same port. The default port that is assigned is 8081. The scripts that run those services are `jini_codeserver.bat` and `run_phoenix.bat` respectively.

Once you see "INFO:.." on each command window, the services are up and running. Then, we need to initialize DCS.

3. Initalize the DCS by running `initOctopus.bat` in the folder `jini\octopus\bin`.

4. Deploy DCC in those machines that are going to be publishers and acquirers by unzipping *DCC.zip*. The root directory is The `octopus`.

5. Run the script `runOctopus.bat` in the folder `octopus\bin`. This will start a GUI.

The DCC has a graphical user interface (GUI). The GUI has a tab pane that gives a choice of being a publisher or an acquirer. A publisher has a choice of publishing the problem of finding a quote, or simulating a portfolio model, or both. Before clicking on the 'Publish Problem' button, one needs to check either one problem or both problems. The 'Publish Problem' button publishes the checked problems into DCS. The 'Acquire Results' button finds all the results available for the publisher and processes those results. The resulting information is displayed on the text area. The 'Remove Problem' button removes one problem at a time that the publisher has published in the DCS.

An acquirer has a choice of solving a problem by clicking on the button 'Acquire Problem.' When the button is clicked, the acquirer gets a problem from the DCS, process it, creates a **ResultsEntry** object, and then publishes to the DCS.

In this Chapter, we discussed the implementation details of the Octopus. In the next Chapter 9, we present our accomplishments, contribution, and how the project can be extended.

9 Further Work

The goal of the project has been to develop a Peer-to-Peer (P2P) Distributed Computing environment, named Octopus, such that we can capitalize unused computer resources to solve computing intensive problems. We have designed, developed, and demonstrated that it is a formidable solution for common business problems by using wired or wireless resources.

Our simple demonstration has shown that the Octopus architecture is a viable framework for P2P Distributed Computing. Though the current demonstration version contains a minimal set of functions, we make a case that this is a viable product for solving business problems and managing corporate information, with additional functionality such as security, clustering, and so on.

The Octopus framework harmonizes devices in the network. Though designed to taking advantage of idle resources in a network, the framework is capable of not only utilizing resources in the network but also a medium for devices to communicate among peers in the network. Since the Octopus architecture addresses issues with distributed computing such as partial failure and latency as described in Chapter 7, it is a framework for guaranteeing at least a solution.

Our vision of a production-ready product is a framework for P2P Distributed Computing that a business can use to solve business problems and manage information. The product may be tailored to a client's specific problem. However, we can package a few universal problems. One such problem is managing personal data such as calendar, emails, notes, and so on, among various devices such as PDA's, cell phones, desktops, and so on. We can provide an out of box solution for personal data synchronization along with the Octopus framework.

We believe the natural extension of the applications of this product besides the financial industry is medical data processing. Medical data processing involves gathering data from medical devices from patients, transmitting and storing those data in

appropriate storage in the network, analyzing those data, and sending the results to appropriate devices such as the PDA of the patient's primary physician for review. The Octopus architecture provides a suitable framework for harmonizing medical devices, using resources, and managing medical data and information efficiently in a network. Therefore, we can provide timely medical services to patients. There is a pressing need to improve current medical processing as 'baby boomers' age and require medical attention. We believe Octopus is a choice computing framework for medical data processing.

If we get funding, what is our strategy to develop a fully-grown product? We need to add functionality such as security, high availability, and extensible API's that we use to add problems such that acquirers can readily process them without a need for a software upgrade.

The first functionality that we add is the security model supported by the Jini 2.0 specification. The security model should be able to authenticate and authorize a code fragment to prevent any malicious code. Besides the authentication and authorization, we distribute a digital certificate to each publisher to publish problems in DCS as a way of preventing unwanted intrusion to the system. Acquirers can use the same digital certificate to authenticate the authenticity of the request. Similarly, the digital certificate is used to authenticate the publishers to allow removing problem and results that belong to them only from the DCS.

We need to create a robust and secure DCS (JavaSpace). We need to be able to cluster DCS for performance, high availability, load balancing, and so on. We use the SUN implementation of the JavaSpace specification, which is a simple implementation. There are a few commercial JavaSpace products such as "GegaSpace" that we take leverage instead of implementing our own JavaSpace. This allows us to provide a secure, robust, and high availability DCS.

Once we have a secure infrastructure, we need to secure information. For example,

the medical information cannot be compromised as regulations such as the HEALTH INSURANCE PORTABILITY AND ACCOUNTABILITY ACT (HIPPA) dictates security guidelines for the medical information. The penalty is high when security is compromised. It is not only the law, but the very existence of a corporation depends on security of the corporation's vital information. Breach of corporate secrecy can be fatal to the corporation. Therefore, securing information is vital to the success of a distributed application. This requires the Octopus to encrypt data and problems. The data encryption also allows us to discriminate acquirers as service providers such that only certain acquirers have expertise or resources for providing services for such problems. Those acquirers have keys to decrypt the data and problems.

We develop an encryption model that creates a hierarchy of security. The acquirers have different levels of security clearance as in an intelligence agency. This allows creating different levels of service providers. The acquirers have keys corresponding to their security clearance. The encryption model, i.e. security clearance, also can be thought of as the abilities of an acquirer.

The Octopus framework needs to encompass mobile devices such as J2ME and Bluetoothe devices. This will allow us to extend service to small mobile devices. This is where medical devices can take advantage of the framework, as those devices may be small and mobile.

True P2P Distributed Computing is only possible when there is a common language with full vocabulary so that acquirers can decipher publishers' requests. Java has standardized programming to an extent. Similarly, XML has provided a standardized data format. Jini has taken one step forward toward standardizing objects in a memory. An example of object standardization is the use of the
net.jini.core.entry.Entry Interface as a requirement for writing objects in a JavaSpace. Such standardization provides a common object language that a community of computers can use for understanding each other's request. Although we have come a long

way in developing common standards, we still lack a common vocabulary for P2P Distributed Computing. We need a language with vocabularies in which we can describe a problem such that the members of community can understand without deploying new version of software. Just as the English language has provided a medium of communication among English speaking communities, we need a language with a rich vocabulary to describe our needs so that a member in the community can understand a request with ease and fulfill the request. Jini has paved a path for such a language; however, we do need to extend Jini language capacity to create a standard language with full vocabulary to describe our needs. That language can provide a true common object language such that all acquirers and publishers alike can parse the information with ease to make pure P2P Distributed Computing a reality.

In conclusion, the Octopus architecture applies to any devices connected in a wired or wireless network. Thus, the Octopus system encompasses mobile and wireless devices along with desktop computers and servers. The strength of the Octopus architecture is that it accepts any hardware capable of running a JVM. We see a huge application of the Octopus architecture in managing information, solving problems, exchanging information among devices whether it is wired or wireless, static, or mobile devices.

Appendix A

Experimental Results

Following tests were conducted in a closed network with four computers. One computer was dedicated as a DCS, and other three can choose to be a publisher, an acquirer, or both. During the experiment, the DCS held between 7 and 12 problems at any given time. We report the following observations.

Test scenario A is to have two publishers to publish two different kinds of problems.

Octopus did not have any problem distributing problems and returning solutions. Octopus was able to distribute Stock Quote as well as Portfolio Simulation problems. Also, Octopus was able to publish results.

Test scenario B is to crash or remove publishers and acquirers in a systematic and repeatable way.

The results show that acquirers were able to get a problem and solve it once on the network. Similarly, publishers were able to publish problem and receive results without any problems.

Test scenario C is to observe if there is any impact on providing solutions to problems when acquirers abruptly disappear.

When an acquirer abruptly disappeared, it did not have an impact on receiving at least one solution. The problem was solved by another member in the network community.

Contents of CD

The following are the items that are included in the CD.

1. A user guide.
2. OctopusDCC-2.zip - Software to install in peer computers.

3. OctopusDCS-2.zip - Software to install JavaSpace and Jini services.

4. Source code.

5. Javadoc.

References

[1] Philip E. Agre. P2P and the Promise of Internet Equality. *Communications of the ACM*, 46(2):39–42, February 2003.

[2] Hari Balakrishnan, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Looking Up Data in P2P Systems. *Communications of the ACM*, 46(2):43–48, February 2003.

[3] David Barkai. Technologies for Sharing and Collaborating on the Net. *Peer-to-Peer Computing, 2001. Proceedings First International Conferece on Peer-to-Peer Computing*, pages 13–28, 2002.

[4] Richard A. Brealey and Stewart C. Myers. *Principles of Corporate Finance.* McGraw-Hill, Inc., New York, NY, 1991.

[5] Intel Corporation. Peer-to-Peer-Enabled Distributed Computing. White Paper/Case Study, Intel Corporation, 2001.

[6] Geoffrey Fox. Peer-to-Peer Networks. *Web Computing*, pages 75–77, May/June 2001.

[7] Eric Freeman, Susanne Hupfer, and Ken Arnold. *JavaSpaces Principles, Patterns, and Practice.* Addison-Wesley, California, 1999.

[8] James Gosling, Bill Joy, and Guy Steele. The Java Language Specification. Java Language Specification, Sun Microsystems, Inc., 901 San Antonio Road, Palo Alto, California 94303 U.S.A., April 1998.

[9] Steven L. Halter. *JavaSpaces Example by Example.* The Sun Microsystems Press, New Jersey, 2002.

[10] John L. Hennessy and David A. Patterson. *Computer Architecture, A Quantitative Approach.* Morgan Kaufmann Publishers, 2003.

[11] Eric Korpela, Dan Werthimer, David Anderson, Jeff Cobb, and Matt Lebofsky. SETI@home: Massively Distributed Computing for SETI. *Computer Society, IEEE.*

[12] Philippe Kruchten. *The Rational Unified Process An Introduction.* Addision-Wesley, 2000.

[13] John Kubiatowicz. Extracting Guarantees from CHAOS. *Communications of the ACM*, 46(2):33–38, February 2003.

[14] Kuldeep Kumar, Clarence Tan, and Ranadhir Ghosh. Using chaotic component and ANN for forecasting exchange rates in foreign currency market. Working Paper, School of IT, Bond University, 2002.

[15] Jintae Lee. An End-User Perspective on File-Sharing Systems. *Communications of the ACM*, 46(2):49–53, February 2003.

[16] J.P. Morgan. RiskMetrics - Technical Document. Risk measurement, J.P. Morgan, Reuters, New York, December 1996.

[17] Manoj Parameswaran, Anjana Susrla, and Andrew B. Whinston. P2P Networking: An Information-Sharing Alternative. *Computer*, 34:31–38, July 2001.

[18] Frank K. Reilly and Keith C. Brown. *Investment Analysis and Portfolio Management.* The Dryden Press, 1997.

[19] Detlef Schoder and Kai Fischbach. Peer-to-Peer Prospects. *Communications of the ACM*, 46(2):27–29, February 2003.

[20] Rudiger Schollmeier. A Definition of Peer-to-Peer Networking for the Classification of Peer-to-Peer Architectures and Applications. *Peer-to-Peer Computing, 2001 Proceedings First International Conference on Peer-to-Peer Computing*, pages 101–102, 2002.

[21] W. T. Sullivan, D. Werthimer, S. Bowyer, J. Cobb, D. Gedye, and D. Anderson. A new major SETI project based on Project Serendip data and 100,000 personal

computers. *Astronomical and Biochemical Origins and the Search for Life in the Universe*, Procedure of Fifth International Conference(161), July 1997.

[22] Sun. Jini Architectural Overview. Technical Report, Sun Microsystems, California, 1999.

[23] Sun. Jini Architecture Specification. Technical Report 1.2, Sun Microsystems, California, December 2001.

[24] Sun. Jini Technology Core Platform Specification. Technical Report 1.2, Sun Microsystems, California, December 2001.

[25] Sun. Java Remote Method Invocation-Distributed Computing for Java. Technical Report, Sun Microsystems, California, March 2002.

[26] Sun. Java Remote Method Invocation Specification. Technical Report 1.8, Sun Microsystems, California, Jan 2002.

[27] Sun. JavaSpaces Service Specification. Technical Report 1.2.1, Sun Microsystems, California, April 2002.

[28] Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall. A Note on Distributed Computing. Technical Report, Sun Microsystems Laboratories, Inc., Mountain View, CA 94043, November 1994.