CompuP2P: A light-weight architecture for Internet computing

by

Varun Sekhri

A thesis submitted to the graduate faculty
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Engineering

Program of Study Committee:
Arun K. Somani, Major Professor
Zhao Zhang
Sarah M. Ryan

Iowa State University
Ames, Iowa
2005

Copyright © Varun Sekhri, 2005. All rights reserved.
Graduate College
Iowa State University

This is to certify that the master's thesis of

Varun Sekhri

has met the thesis requirements of Iowa State University

Signatures have been redacted for privacy
TABLE OF CONTENTS

LIST OF TABLES .................................................. vi
LIST OF FIGURES .................................................. vii
ABSTRACT ................................................................. ix

CHAPTER 1. Introduction ........................................... 1
  1.1 Introduction .................................................. 1
  1.2 Contrast with Grid and Public Resource Computing .......... 2
  1.3 Contribution of this thesis .................................. 3
  1.4 Organization of the thesis .................................. 4

CHAPTER 2. CompuP2P: An Overview ............................. 5
  2.1 Computing Resources layer .................................. 6
  2.2 Resource Trading layer ...................................... 7
  2.3 Service layer ................................................ 8
  2.4 CompuP2P Usage ............................................ 8

CHAPTER 3. Resource Trading .................................... 11
  3.1 Constructing Resource Markets ............................... 12
    3.1.1 Single Overlay Scheme ................................. 13
    3.1.2 Processor Overlay Scheme ............................. 14
  3.2 Resource Pricing ........................................... 16
    3.2.1 Overview ............................................... 16
    3.2.2 Providing Incentives to Sellers ....................... 17
CHAPTER 4. Prototype Implementation of CompuP2P  21

4.1 Implementation Details  22
  4.1.1 Programming Language  22
  4.1.2 Node communication  23
  4.1.3 Measuring CPU Load  23
  4.1.4 Heterogeneity  24

4.2 Hybrid Scheme for Constructing Resource Markets  24

4.3 User Interface  26
  4.3.1 Usage Policy Tab  26
  4.3.2 Resource Sharing  28
  4.3.3 Task Submission  29
  4.3.4 File Storage  30

4.4 Remote Execution API  31

4.5 Bootstrap Server  32

4.6 CompuP2P execution on multiple nodes  33

CHAPTER 5. Fault Tolerance Computing  35

5.1 Serverless Checkpointing  35

5.2 Stabilization of CompuP2P  37
  5.2.1 Reachability  37
  5.2.2 Node Failures  38
  5.2.3 Node Joins  40

5.3 Experiments to test Fault Tolerance in CompuP2P  40

CHAPTER 6. Applications and Overhead  43

6.1 PlanetLab Testbed and Experimentation  43

6.2 CompuP2P Overhead  44
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAPTER 7. Comparison with Related Projects and Conclusion</td>
<td>52</td>
</tr>
<tr>
<td>7.1 Related Distributed Computing Projects</td>
<td>52</td>
</tr>
<tr>
<td>7.2 Conclusion and Future Work</td>
<td>53</td>
</tr>
<tr>
<td>APPENDIX</td>
<td>55</td>
</tr>
<tr>
<td>BIBLIOGRAPHY</td>
<td>57</td>
</tr>
<tr>
<td>ACKNOWLEDGEMENTS</td>
<td>61</td>
</tr>
</tbody>
</table>
LIST OF TABLES

Table 4.1  Comparison of market creation schemes. N is the number of nodes and M is number of MO's in system. .................. 25
Table 5.1  Time Period for various routines in CompuP2P. .............. 43
Table 6.1  Performance Comparison of SGA and PGA. .................... 47
Table 6.2  Execution of nQueens problem on PlanetLab nodes ........... 50
Table A.1  Prisoner's Dilemma ............................................. 56
LIST OF FIGURES

Figure 2.1  CompuP2P: System architecture .......................... 6
Figure 2.2  Chord: An identifier circle consisting of three nodes 0, 3, 5 and three keys 3, 4, 6. ........................................... 7
Figure 2.3  A conceptual view of CompuP2P ............................ 9
Figure 3.1  Creation of markets for CPU cycles in CompuP2P .......... 13
Figure 3.2  Processor overlay schema using the CPU capacity values given in Fig. 3.1. .......................................................... 15
Figure 4.1  Screen snapshot of Usage Policy tab. ...................... 26
Figure 4.2  Screen snapshot of Resource Trading tab. ................ 27
Figure 4.3  XML-based task file .......................................... 28
Figure 4.4  Screen snapshot of AdminServer. ......................... 31
Figure 4.5  User constraints specified in XML file. .................... 32
Figure 5.1  Serverless Checkpointing in CompuP2P .................... 34
Figure 5.2  Anonymous lookup protocol steps .......................... 35
Figure 5.3  Listing of a seller at $m + 1$ $MO$ nodes .................. 39
Figure 5.4  Replication of seller entries when primary $MO$ node fails. .... 40
Figure 5.5  Replication of seller entries when a backup $MO$ node fails. ... 41
Figure 6.1  Algorithm for solving N-Queens Problem .................. 47
Figure 6.2  Algorithm for solving N-Queens Problem on multiple processors 48
Figure 6.3 Comparison of parallel and sequential execution of N-Queens problem. 48
ABSTRACT

Internet computing is emerging as an important new paradigm in which resource intensive computing is integrated over Internet-scale networks. Over these large networks, different users and organizations have potential to share their computing resources, and computations can take place in a distributed fashion. In such an environment, a framework is needed in which the resource providers are given incentives to share their resources. In this research we propose CompuP2P, which is a light-weight architecture for enabling Internet computing. It uses peer-to-peer networks for sharing of computing resources. CompuP2P creates dynamic markets of network accessible computing resources, such as processing power, memory storage, disk space, etc., in a completely distributed, scalable, and fault-tolerant manner. We discuss the system architecture, functionality, and applications of the proposed CompuP2P architecture.

We have implemented a Java based prototype of CompuP2P. We ran several algorithms with coarse grained parallelism on CompuP2P. Our results show that the system is light-weight and can provide almost a perfect speedup for applications that contain several independent compute-intensive tasks.
CHAPTER 1. Introduction

1.1 Introduction

Internet computing is a distributed computing paradigm that uses Internet as a single large virtual computer. Internet computing promises to fulfill the vision of “anytime” “anywhere” computing. Applications benefiting from this paradigm range from simple data sharing to ones using Internet as a processing engine for large-scale data storage and distributed task execution. Internet computing is challenging to realize primarily because of its sheer size and open un-trusted environment. In the last decade the concept of Internet computing has been revolutionized due to applications such as file sharing developed around the peer-to-peer (P2P) paradigm. We believe that the P2P paradigm has the potential to serve as a platform for developing several “killer-apps” for making true Internet computing a reality (and also affordable). Some interesting applications are as follows:

1. Allowing processing limited device, such as wireless clients, to distribute their processing requirements to other machines in the network,

2. Utilizing the storage capacity of virtually millions of machines connected to Internet, etc.

Peer-to-peer (P2P) networks [1, 2, 3, 4, 5] are flexible distributed systems that allow nodes (also called peers) to act as both clients and servers and provide services to each other. P2P systems evolve from client-server systems by removing the asymmetry in
roles: a client is also a server that allows access to its resources. There is no notion of a client or a server, and every node in the network becomes a peer. P2P is a powerful emerging networking paradigm that permits sharing of virtually unlimited data and computational resources in a completely distributed, fault-tolerant, scalable, and flexible manner. To enable large-scale resource sharing, a market economy based framework is needed so that computing resources are buyable and sellable on demand in short periods of time. This would give incentive to individuals or large organizations to share their compute resources.

1.2 Contrast with Grid and Public Resource Computing

Internet computing along with grid computing and public resource computing share the goal of better utilizing existing computing resources. However, there are profound differences among the three paradigms.

Grid computing [7] involves organizationally-owned resources: supercomputers, clusters, and PCs owned by universities, research labs, and companies. These resources are centrally managed by IT professionals, are powered on most of the time, and are connected by high bandwidth network links. Malicious behavior, such as intentional falsification of results are handled outside the system, e.g. by using a legal system.

Public resource computing [6] involves an asymmetric relationship between projects and participants. Projects are typically small academic research groups with limited computer resources, expertise, and manpower. Most participants are general Internet users with PCs, workstations, etc., with low bandwidth connectivity to the Internet. The computers are frequently turned off or disconnected from the Internet. Participants contribute their resources either out of altruism or they receive suitable “credit” for doing so. Projects have no control over participants, and cannot prevent malicious behavior.

In contrast, the Internet computing paradigm aims to create a single large hetero-
geneous pool of computing resources into which users can tap into to carry out their tasks. Here, users can include enterprises, research groups, or even individual home PC owners. The system is typically large with thousands or even millions of users. Network connectivity, as in public resource computing, is sporadic. There is no centralized entity that controls the behavior of individual users, and thus users can be expected to behave selfishly (and even maliciously). Due to the large-scale, dynamism, openness, and heterogeneity of these systems, building a platform for Internet computing presents several unique and interesting research challenges.

1.3 Contribution of this thesis

Contributions of this thesis are as follows:

- The first contribution of this thesis is CompuP2P, a light-weight architecture for Internet Computing. CompuP2P uses peer-to-peer networks for sharing of computing resources. It creates dynamic markets of network accessible computing resources, such as processing power, memory storage, disk space, etc., in a completely distributed, scalable, and fault-tolerant manner. CompuP2P uses ideas from game theory [8] and microeconomics [10] to devise incentive-based schemes for motivating users to share their computing resources with each other.

- The second contribution of this thesis is identifying and building fault tolerance mechanisms in CompuP2P. We propose a new scheme for checkpointing a computing node’s processing state, server-less checkpointing. In this scheme, nodes that store the checkpoint data are determined on-the-fly based on their available disk space.

- The third contribution of this thesis is a Java based implementation of CompuP2P architecture and the related fault tolerance mechanisms. We tested several algo-
rithms with coarse grained parallelism on CompuP2P. Our results show that the system is light-weight, and can provide almost perfect speedup for applications that contain several independent compute-intensive tasks.

1.4 Organization of the thesis

Chapter 2 gives an overview of CompuP2P system architecture and briefly explains how CompuP2P works. Chapter 3 describes the protocols for market creation and resource pricing in CompuP2P. In Chapter 4, we present our prototype implementation including various features built in CompuP2P. Chapter 5 introduces server-less checkpointing and describes steps taken in the implementation of CompuP2P in the event of node failures. In Chapter 6, we discuss some applications which are amenable for large scale parallel computing and can use idle computing resources made available by CompuP2P. In section 6.2, we evaluate overhead incurred by nodes in the system for message communication or state maintenance. In Chapter 7, we compare CompuP2P with other similar distributed computing projects. Finally, in Section 7.2 we conclude this thesis and mention some future work which can complement the work presented in this thesis.
CHAPTER 2. CompuP2P: An Overview

CompuP2P uses a peer-to-peer architecture for creating markets for trading of computing resources such as CPU cycles, disk space, etc. CompuP2P create different markets for different amounts of a computing resource, referred to as a commodity. Nodes that are responsible for running different commodity markets are termed as “market owners” (MOs). MOs are dynamically re-assigned as nodes leave and join the network. A MO does the job of a matchmaker between sellers and buyers, and maintain information about the sellers. This information can include available compute power and/or disk space, operating system type and version, platform type, price requirement, etc. Upon receiving a request from a client, the MO returns the information about the seller that best meets the client’s requirements. A Chord-based protocol is used for market creation and lookup, and with high probability both sellers and buyers of a commodity converge on the same market, i.e., both sellers and buyers contact the same MO that is responsible for running the market for that commodity. It must be noted that a single physical node can be a MO for various commodities.

CompuP2P is designed to take into account users’ selfishness, and uses ideas from game theory and microeconomics for pricing of computing resources. CompuP2P allows users to define their policies regarding what, when, how, and by whom their resources can be used. Moreover, it allows users to specify their task requirements while accessing the system resources. For example, a user can specify the timeliness and reliability requirements regarding the received results.

Figure 2.1 depicts the layers constituting the CompuP2P architecture. The function-
The functionality of these layers is explained in the following subsections.

2.1 Computing Resources layer

This layer refers to various distributed resources, such as compute power, disk space, files, etc., that exist in any large Internet-scale system. These resources belong to different nodes that are part of the underlying P2P network. In our prototype implementation of CompuP2P, nodes in the system are organized in a Chord ring [4].

Chord is a distributed lookup protocol to efficiently locate a node that stores a particular resource. It provides support for just one operation - given a key, it maps the key onto a node. In Chord, each user joining the network is assigned an $m$-bit identifier (or ID), obtained by hashing the IP address of the node. These nodes are arranged in an identifier circle, modulo $2^m$, according to the assigned IDs.

The keys are also mapped into this ID space, by hashing them to $m$-bit key IDs. A key $k$ is assigned to the first node whose identifier is equal to or follows (the ID of) $k$ in the identifier space. This node is called the successor node of key $k$, denoted by...
successor\( (k) \). If identifiers are represented as a circle of numbers from 0 to \( 2^m - 1 \), then 
successor\( (k) \) is the first node clockwise from \( k \).

Figure 2.2 Chord: An identifier circle consisting of three nodes 0, 3, 5 and three keys 3, 4, 6.

Figure 2.2 shows an identifier circle with \( m=3 \). The circle has three nodes 0, 3, and 5. The successor of key identifier 3 is node 3, so key 3 is located at node 3. Similarly, key 4 would be located at node 5 and key 6 at node 0.

A key can be looked up via \( O(\log N) \) messages, where \( N \) is the number of nodes in the system. For a detailed explanation of Chord reader is referred to [4].

Although, we have used Chord as the underlying P2P protocol, the architecture of CompuP2P is generalized enough to be built on top of other structured P2P networks, such as CAN [5].

2.2 Resource Trading layer

Functionality of Resource Trading layer is further divided into three sub-layers:

- Market lookup protocol: It ensures that sellers and buyers looking to trade a commodity converge on the same market.
• Resource pricing protocol: The pricing mechanism ensures that both sellers and MOs are suitably compensated for the service they provide to clients.

• Dynamic market creation protocol: It is used for selecting nodes that act as MOs for specific commodities. The protocol is robust against MOs failing and new nodes joining the system.

This layer is described in detail in Chapter 3.

2.3 Service layer

The service layer accepts service requests from a user. A service can be a computation task or a data storage request. A computation task is submitted by a user in the form of a task file. The task file is parsed and appropriate computing nodes in the network are determined that can execute the associated sub-tasks. The service layer allows a user to specify reliability and timeliness requirements on the result of computation, while accessing the system resources. Moreover, a user interested in backing up the local data can also request the service layer to search for appropriate storage nodes in the network. The data is usually replicated at multiple remote nodes and is stored in either plain-text or encrypted format.

2.4 CompuP2P Usage

Figure 2.3 depicts a conceptual view of the operation of a CompuP2P system. Next, we briefly describe the steps taken by a user to carry out a distributed computation.

1. The user submits a task file to the service layer. The task file specifies for each sub-task various attributes, such as input parameters, CPU cycles required, maximum offered price for successful execution, etc.
2. The service layer parses the task file and queries the resource trading layer for the appropriate seller nodes.

3. The resource trading layer looks up the market(s) that trade in resources required for the successful completion of the sub-task(s).

4. The \textit{MO} node is queried about the available sellers. The \textit{MO} acts as a matchmaker for finding the best seller, for example, the seller that offers the needed compute power at minimum cost.

5. The resource trading layer returns the information, like the IP address of a selected seller, to the service layer.

6. The service layer submits the sub-task to the seller. Depending on a user’s requirements, the service layer also provide various fault-tolerance features such as checkpointing and replicated computing.
7. Finally, the user is notified of the execution results with the output(s) being stored in appropriate file(s), as specified by the user in the task file for the sub-task.
CHAPTER 3. Resource Trading

At the heart of CompuP2P is its resource trading layer. This layer is responsible for creation and management of markets. For concreteness, here compute power has been used as the resource under consideration. However, the proposed mechanisms for resource trading are equally applicable to other resource types, such as disk space, memory, bandwidth, etc. For details of the work and proofs of theorems presented in this chapter reader is referred to [26].

Each node based on its current and past load estimates the average number of resources that would remain idle in future, for example on Unix platform a user can use commands “top” and “uptime”. Suppose a node determines that it has \( C \) cycles/sec available for the next \( T \) time units (where \( T \) is some large enough time period) that it can provide or make available to others for processing. In case some other resource, say disk space, is under consideration then another appropriate unit like \( G \) gigabytes for \( T \) time units can be used. It must be noted that the same number of CPU cycles/sec might represent different amounts of compute power for different nodes. This might happen if nodes have different hardware and/or software configurations. Here unit of cycles/sec is used to represent normalized equivalent amounts of compute power at different nodes in a heterogeneous system.

Once the amount of idle compute power has been estimated, the next step is to determine how to sell them. Moreover, buyers needing extra compute power should be able to locate the right sellers and purchase the needed CPU cycles from them. The related and equally important issue is how the sellers should price their CPU cycles in
order to maximize their profits. These issues are addressed in the following sections.

### 3.1 Constructing Resource Markets

Since different nodes have different amounts of compute power to sell and purchase, it is necessary to create suitable markets to permit buyers and sellers to come together and trade the amount of compute power they require. For a buyer to sequentially search the entire network for the best available deal is a very time consuming and expensive operation. Also, selecting one node, say the successor of Chord ID zero, for trading all available compute power in the network is not a good idea either. This is because relying on one node can lead to extreme scalability, fault-tolerance, and security problems.

For efficient creation and lookup of compute power markets, two schemes have been proposed in [26] that attempt to uniformly distribute the location of and responsibility for maintaining those markets across the network. Both the schemes use Chord for market assignment and lookup, however, they differ from each other in the overhead involved and the manner in which nodes are selected for running markets for various commodities. The term *commodity* as used here represents a range of idle CPU cycles/sec values. Each market deals in only one type of commodity (i.e., homogeneous markets). A single physical node may be responsible, i.e., be a market owner (*MO*), for more than one market.

Figure 3.1 depicts how nodes with different values of idle compute power $C$ join different markets. Here, for simplicity of discussion, $C$ represents a discrete value, but in actual practice it refers to a well-defined range of values within which a node's idle processing capacity can lie. Thus, nodes with different but close enough idle processing capacities trade in the same market.

Two schemes for the creation of compute power markets are described below.
3.1.1 Single Overlay Scheme

In this scheme, the value $C$ computed by a seller acts as the Chord ID for locating the corresponding compute power market. The successor node of Chord ID $C$ is assigned the responsibility for maintaining the market for that particular idle compute power. It is possible that several compute power values map to a single node and then that node is responsible for running different markets, all dealing in different commodities.

This scheme is very simple to implement and involves not much additional overhead. Compute power markets are searched using the normal Chord lookup protocol. In other words, if a node needs to purchase $x$ cycles/sec, it simply looks up for the market maintained by the successor of Chord ID $x$. The drawback of this scheme is that if the idle compute power values in the network happen to be in a very narrow range, then most of the markets would map to only a very few distinct physical nodes. Those nodes can then become the bottleneck and degrade the system performance. Moreover, search for a suitable market by a buyer might potentially require several attempts. In each attempt the amount of compute power searched for is successively increased, until a desired seller with adequate capacity is discovered.
3.1.2 Processor Overlay Scheme

In order to more uniformly distribute the responsibility for running the compute power markets and to bound the search time for an appropriate seller, an additional overlay can be maintained that keeps information about available idle compute power at different sellers in the network. All MOs, which are responsible for various commodities, constitute this Chord-based overlay network. The total ID space of this new overlay is equal to the maximum amount of compute power that may possibly be available on any single node and is upper-bounded by $2^c - 1$, where $c$ is a constant and represents the number of bits used to represent the maximum value of idle CPU cycles/sec. Here the value of $c$ is assumed to be large enough to represent the idle processing power of even a very large computer system.

The process of selecting a MO for a commodity is illustrated in Figure 3.2. It depicts an existing Chord network comprising of all the nodes, and $m$ is the Chord ID size in terms of the number of bits. A node on determining its value for $C$ applies a hash function to $C$ to find the corresponding Chord ID ($= \text{hash}(C)$, a value between 0 and $2^m - 1$. The successor node of $\text{hash}(C)$ is then the MO for the market trading in commodity $C$. The various MOs defined in this manner then together form another overlay network, called the processor overlay, which has ID space from 0 to $2^c - 1$. The ID of a MO in this new overlay network is simply the value $C$ whose hash value was mapped to it in the initial Chord network. Stated otherwise, the ID of a MO in the processor overlay network, called CPU Market ID (CMID), is the number of CPU cycles/sec that are being sold in its market.

It must be noted that in the above description, it is possible that a single node in the initial overlay network is the MO for several different markets, causing it to have multiple CMIDs assigned to it in the processor overlay network. Each CMID value is represented by a different node in the processor overlay, as shown in Figure 3.2. MO's
(i.e., nodes comprising the processor overlay) periodically send out a broadcast message identifying themselves to nodes in the network. Each node needs to store information about a single MO (or at most few for fault-tolerance) to be able to perform lookups in processor overlay.

![Diagram](image)

**Figure 3.2** Processor overlay schema using the CPU capacity values given in Fig. 3.1.

The lookup in processor overlay, requires $\frac{1}{2}(\log M)$ steps on average, where $M$ is the number of different markets. Moreover, nodes store $O(\log M)$ routing information to support the Chord protocol.

The search mechanism for the compute power in processor overlay is performed based on the number of CPU cycles/sec (which acts as the lookup key) that a client requires for processing. The client first contacts any of the known MOs and forwards the lookup request to it. The selected MO searches for an appropriate market for the desired compute power in the processor overlay network. The lookup process finally returns the IP address of the MO that runs the market for that compute power or the nearest higher compute power value available in the network. For example, if only two compute power markets (with commodity values $b$ and $c$) exist in the network, and a client desires $a$ (where $a < b < c$), then the above mechanism returns market for $b$ instead of $c$. The MO is then contacted to obtain information about the sellers listed in the market.
3.2 Resource Pricing

3.2.1 Overview

Pricing is non-trivial when there are either multiple at par sellers from a buyer's point of view or when a buyer is trying to minimize its cost of processing (again assuming multiple sellers). Utilizing the model that a transaction involving the trading of compute power can be modeled as a one-shot game and using the results from game theory and microeconomics (the classical Prisoner's dilemma problem [8] and Bertrand oligopoly [10], respectively), it can be concluded that long-term collusion among compute power sellers (and MO) is unlikely to occur. In one-shot Prisoner's dilemma game, non-cooperation is the only unique Nash equilibrium strategy for the players. In fact, the model of Bertrand oligopoly suggests that sellers (irrespective of their number) would not be able to charge more than their marginal costs for selling their resources (see [8] for a game-theoretic derivation of this result). In Bertrand oligopoly sellers strategy is to set "prices" (as opposed to "outputs" in Cournot oligopoly), and is thus more reasonable to assume in the context of CompuP2P. In CompuP2P all the sellers in a market sell the same amount of a computing resource. As a consequence, sellers, irrespective of how many there are in a market, in CompuP2P set prices equal to their marginal costs only. The marginal cost of providing a computing resource can include among other things - listing price, bandwidth cost for message exchange, etc., and is represented by $MC_i$ for a node, $i$. More explanation on game theory is provided in Appendix A.

One-shot model of a compute power transaction is reasonable to assume, since once a seller sells its compute power, it de-lists itself from the market and perhaps move to another market for selling its remaining compute power, if available. Moreover, in a dynamic system, where nodes continually join and leave the network, it is difficult to keep track of nodes that do not fulfill their collusion agreements. Thus nodes are not likely to be penalized based on their past behavior.
3.2.2 Providing Incentives to Sellers

Since the best pricing strategy for sellers is to charge equal to their marginal costs, it results in zero profits for them. Therefore, sellers would not be motivated to sell their computing resources unless some other incentive mechanisms are devised for them. Below two such strategies depending on whether fixed or variable listing pricing is used to compensate a MO are described.

- **Strategy For Fixed Listing Pricing.** If fixed listing pricing is possible, then a MO has no incentive to cheat and thus we can use the technique employed in Vickrey auction [9]. A seller, when it joins a market, provides its marginal cost information to the MO. A buyer, looking to minimize its cost, selects the seller with the least marginal cost, but the amount it has to pay to the seller is equal to the second lowest marginal cost value listed in the market. This selection scheme is called reverse Vickrey auction.

The above strategy provides non-zero profit to the selected seller and ensure that sellers state their correct marginal costs to the MO (see [9] for the truth-eliciting property of Vickrey auction). The strategy is also inherently secure because even if sellers learn about the posted marginal costs, they cannot take undue advantage of that information to post a lower marginal cost than their actual values. To understand this, consider the following simple example.

**Example:** Suppose seller A has the marginal cost \(MC_A\) of 5 and the lowest marginal cost among all the sellers different from A \(= MC_A^{-1}\) is 4. If A hides its true MC and posts it as 3 in order to get selected, its actual payoff would be \((MC_A^{-1} - MC_A)\) or 4-5 = -1, i.e., it would suffer a loss of -1. Thus, it can be seen that the only rational strategy for a seller is to post its correct MC. In this incentive scheme, a seller selected for processing makes a profit of \((MC^{-1} - MC)\).
• **Strategy For Variable Listing Pricing.** If variable listing pricing is being used, the above scheme based on Vickrey auction cannot be employed. This is because Vickrey auction is designed to be used by non-selfish auctioneers (here MO is the auctioneer), whose goals are to maximize system efficiency as opposed to personal gains. Whereas, in variable listing pricing, a MO has incentive to behave selfishly to maximize its profits. For the case of fixed listing pricing this selfishness was not a problem, since the payoff that a MO received was fixed. But if the payoff that a MO receives is dependent on a transaction outcome, then it has incentive to cheat. To understand how a MO may cheat consider the following example.

**Example:** Let us say, a MO receives 10 percent of a transaction value from the sellers. Suppose there are three sellers, A, B, and C currently listed in the market. The marginal costs of A, B, and C are 100, 200, and 300, respectively. If a buyer now makes a request for the lowest cost supplier then the MO has incentive to report C as the lowest cost supplier, instead of A. This is because by doing so the MO earns a profit of 30 (=300*10/100) instead of 10 (=100*10/100). Even if Vickrey auction is used, the MO has incentive to report 200 and 300, instead of 100 and 200 as the lowest and second lowest cost values, respectively, to the buyer.

In order to deal with the selfish MO problem, we propose a *max-min payoff* strategy. This strategy makes the payoff to a seller and MO complementary to each other, i.e., if the seller receives a high payoff then the MO receives a low payoff, and vice versa. We define the following simple model for this strategy. Let there be S sellers in a market, represented by 1, 2, …, S, such that \( MC_i < MC_{i+1} \) for all \( 1 \leq i \leq S - 1 \). The sellers are not aware of each other (and of the buyers) and only know their own marginal costs, which they truthfully report to the MO. Buyers are also completely unaware about the sellers that are listed in the market and rely on the MO to give them information about the lowest cost supplier.
The payoffs to the MO and the selected seller by the buyer under max-min payoff strategy (based on the marginal cost values that a buyer receive from the MO) are as follows.

\[ \text{Payoff}_{MO} = \frac{(MC' - MC')}{(MC')^2} + \delta \]

\[ \text{Payoff}_{seller} = MC' + 1 \]  

(3.1)

\(MC'_i\) and \(MC'_S\) in the above equation refer to the marginal cost values of the lowest and highest cost supplier, respectively, as reported by the MO to the buyer. Note that a MO can manipulate the reported values if doing so increases its payoff. Also, \(\delta\) is a fixed payoff that a MO receives from a buyer irrespective of the marginal costs of the sellers. Therefore, the max-min payoff strategy can be considered to implement a hybrid listing pricing that has features of both fixed as well as variable listing pricing.

The above payoff values guarantee that the total cost to the buyer is bounded, and the best strategy for the MO is to return the lowest cost supplier only. This can be formalized in the form of the following lemma (for proof see [26]):

**Lemma 1.** Assuming one-shot model of compute power transactions, the payoff strategy in Equation 3.1 guarantees the following.

a) The lowest cost supplier is always selected.

b) The payoff received by the selected seller covers its marginal cost of providing the service.

c) The total cost to the buyer is bounded.

d) The payoff to the MO is variable depending on the dynamics of a market, specifically, it depends on the marginal costs of the sellers listed in the market.
In the above it has been assumed that a MO serve the buyers in the order in which it receive requests from them. Moreover, once a seller has been selected for processing, it de-lists itself from the market, and joins some other market if it has sufficient compute power remaining.
CHAPTER 4. Prototype Implementation of CompuP2P

We have implemented a Java-based prototype of the proposed CompuP2P architecture for sharing of computing resources, and have deployed it in our lab for running compute intensive simulations. In this chapter we explain our prototype implementation of CompuP2P.

4.1 Implementation Details

The symmetric nature of P2P software makes such software systems harder to write than client-server systems. Since there is no notion of clients and servers here, a P2P programmer has to cope up with the types of errors which are encountered only at servers in client-server systems. In this section we will discuss the various problems and design issues faced by us in developing CompuP2P.

4.1.1 Programming Language

Our choice of programming language depended on several factors:

- platform independence,
- object serialization support,
- multithreaded program support, and
- GUI support, with consistent user experience (look and feel) across different platforms.
The above factors made Java the best choice to implement CompuP2P. It provides a rich set of APIs and allows a program to run efficiently on different platforms. The system code is compact (6K lines of Java Code) mostly due to our heavy use of advanced features of Java API (serialization, GUI, class loading etc.). Currently, we have tested CompuP2P on Sun Solaris and Linux Platform.

4.1.2 Node communication

In P2P systems, nodes should be able to connect and access resources from its peers. To facilitate node communication we used Remote Procedure Calls (RPC) since it is easier and intuitive to understand and implement than other methods like message passing. There are many libraries which support making RPCs for example Java RMI (Remote Method Invocation)[23].

We have implemented this feature using XML-RPC [20]. XML-RPC is an excellent tool for establishing wide variety of connections between computers. XML-RPC is far more light-weight than RMI, passing only parameters rather than objects. It hides the network socket programming from the programmer, and allows to implement the remote procedure call (RPC) approach while taking advantage of existing HTTP tools and infrastructure. We have used XML-RPC library provided by the Apache organization [21].

4.1.3 Measuring CPU Load

In CompuP2P, the computing resources are made buyable and sellable in the market. Our goal was to measure average CPU load accurately and present the result to the user which would allow the user to forecast load on the system and be able to determine how much computing power it can make available to peers in the network. On Linux platform, we read the CPU load statistics from /proc/stat file (the “top” command also reads information from this file). On Solaris platform, we read the relevant output from
the 'iostat -c' command. Currently, this feature is not supported on Windows platform because there is no such file or Java API which can provide information. This feature can be implemented by writing a DLL (Dynamical Loadable Library) using Java Native Interface (JNI). But writing DLLs and the precise steps to implement this feature using JNI varies between different Java environments and versions which makes it very difficult to run the same implementation on different computers.

4.1.4 Heterogeneity

Different nodes in CompuP2P may be running on different operating system and processor. A buyer might have specific needs for the environment in which the task submitted should be executed. To solve this heterogeneity problem among nodes, in CompuP2P, MO acts as a matchmaker. MO stores the detailed platform description of all the sellers in the market. This description includes information such as OS type, OS version, processor configuration, etc. On receiving a request from a buyer, the MO selects the seller that not only has the lowest cost, but also meets the platform specifications as desired by a client for its sub-tasks.

4.2 Hybrid Scheme for Constructing Resource Markets

We discussed two methods in Chapter 3 for constructing resource markets. However, while implementing we observed following drawbacks with these schemes:

- As discussed in section 3.1.1, the single overlay scheme suffers with the drawback of mapping most of the markets to only few distinct physical nodes. Searching for a suitable market might also require several attempts.

- The processor overlay scheme, discussed in 3.1.2, suffers with the drawback of overhead as an additional overlay of market owners is maintained to bound the
search time for an appropriate seller and to uniformly distribute the responsibility for running the market for compute power.

- In both the schemes it is assumed that the seller and the buyer will be able to state the computing cycles per second which they want to sell or buy respectively. To measure and quantify computing cycles is a difficult task.

We propose a new hybrid approach to construct resource markets which is also easy to implement. We measure the CPU load by the method discussed in 4.1.3. This CPU load is reported on a scale of hundred, say \( L \). Thus the CPU idle time, say \( I \), is given by

\[
I = 100 - L
\] (4.1)

Each node that is willing to share its computing cycles determines its value for \( I \). A hash function is applied on \( I \) to find the corresponding Chord ID (= hash(\( I \))). The successor node of hash(\( I \)) becomes the MO for market trading. Note that value \( I \) can change very frequently on a desktop machine but to provide stability in the system the node is listed in a new market, after removing itself from the previous MO, only when the difference between the new value and old value of \( I \) is more than 10.

To search for a seller in the network, a buyer estimates the idle compute power required on the scale of hundred (which acts as the lookup key), say \( R \). Successor node of hash(\( R \)) will be the MO node for the buyer. IP address of a seller listed on the selected MO with asked idle capacity (or nearest higher idle capacity), lowest cost and satisfying other constraints specified by the seller is then returned to the buyer.

In this approach a hash of idle capacity is taken like Processor Overlay Scheme and there is no need for maintaining a separate overlay for market owners like in the Single Overlay Scheme. Thus we justly term this approach as a hybrid approach for constructing the computing resource markets. In Table 4.1, we compare the three schemes.
<table>
<thead>
<tr>
<th></th>
<th>Single Overlay Scheme</th>
<th>Processor Overlay Scheme</th>
<th>Hybrid Scheme</th>
</tr>
</thead>
<tbody>
<tr>
<td>Messages Exchanged for seller lookup</td>
<td>$O(\log N)$</td>
<td>$O(\log M), M \ll N$</td>
<td>$O(\log N)$</td>
</tr>
<tr>
<td>Additional Overlay</td>
<td>None</td>
<td>MO's are arranged in new overlay.</td>
<td>None</td>
</tr>
<tr>
<td>Message Overhead</td>
<td>None</td>
<td>MO's broadcast messages identifying themselves.</td>
<td>None</td>
</tr>
<tr>
<td>Node failures</td>
<td>If MO fails, sellers have to relist</td>
<td>Seller can be found in the processor overlay.</td>
<td>Replication (discussed in Section 5.2.2) of MO's ensures availability of seller in the system.</td>
</tr>
</tbody>
</table>

Table 4.1 Comparison of market creation schemes. $N$ is the number of nodes and $M$ is number of MO's in system.

### 4.3 User Interface

Screen-snapshots of the implemented CompuP2P prototype, as it appears to a user, are shown in Figures 4.1 and 4.2.

#### 4.3.1 Usage Policy Tab

The first tab, "Usage Policy", allows a user to specify the usage constraints on the local shared resources. It is divided in four sections:

- **CPU Usage Policy**: User can specify cost of compute power being shared and the CPU load levels (in percentage) beyond which the node is not allowed to share its compute power. Two sliders have been provided. The first slider allows user to specify that compute power is shared if and only if the current CPU load is less than the specified value. Using the second slider a user can specify a maximum CPU load for sharing computing cycles. If the CPU cycles are being shared and the CPU load exceed the specified maximum value then the system delists the seller (user) from the market. To prevent a task from running forever, user can
impose the maximum allowable run time on tasks received for execution. User can also specify the cost of the resource being shared.

- **Memory Usage Policy**: User can specify name of the shared directory, memory space shared in kilobytes and cost of the memory per kilobyte.

- **Time of Day Constraints** By default the compute power is shared at all times during the day. User can specify specific times of the day when compute power can be shared. For example, one can specify that compute power can be shared only during night time when the machine is mostly unutilized.
• **IP Blocking** User can limit access to the machine by specifying the IP addresses of nodes that are not permitted to utilize the shared resources.

### 4.3.2 Resource Sharing

![Figure 4.2 Screen snapshot of Resource Trading tab.](image)

The second tab, "Resource Sharing", is divided into two components. In the first component user specifies whether compute power and disk storage are shareable or not (usage policies described above are consulted before the resources are actually made shareable). The second component lets the user advertise (share) files, which can be downloaded by others in the network. Similarly, a user can search files in the network.
by using the file name. As a part of our future work, if there are more than one files by
the same name being shared by different peers, then the user will be told about the IP
addresses where the files are located and can be downloaded from.

4.3.3 Task Submission

The task file, in CompuP2P, is specified in XML format. It is supplied to the service
layer via the “Task Submission” tab. The task file contains a description of various
sub-tasks (a given task is assumed to be broken into several independent sub-tasks) that
need to be solved. A sample task file is shown in Figure 4.3. For each sub-task, the
following information is included.

```
<?xml version="1.0" ?>
<TaskFile>
  <SubTask>
    <CpuCycles>10</CpuCycles>
    <Price>23</Price>
    <Chkpt>false</Chkpt>
    <ChkptSize>0</ChkptSize>
    <File>java nQueens</File>
    <InputParameters>
      <lCount>4</lCount>
      <Param>14</Param>
      <Param>1</Param>
      <Param>14</Param>
      <Param>res1.out</Param>
    </InputParameters>
    <OutputFiles>
      <OCount>1</OCount>
      <OutputFile>res1.out</OutputFile>
    </OutputFiles>
    <Config>
      <OS>Linux</OS>
      <Processor>Intel</Processor>
    </Config>
  </SubTask>
</TaskFile>
```

Figure 4.3 XML-based task file

- **Code ID** (or name) of the executable file for the sub-task. The executable file is
either locally available or can be searched for and downloaded just as other normal
data using code ID (or name) as the key. A sub-task is executed at a single node and thus define the level of granularity at which parallelism can be achieved.

- Names of input and output files to be used. If the input files are not available with a computing node, they can be searched for using the Chord lookup protocol.

- Estimated amount of compute power required.

- User's budget, i.e., the maximum amount of reward that a user can give in order to get the sub-task successfully executed.

- An indication whether the sub-task is to be periodically checkpointed or not, and the estimated size of checkpoint data.

- Platform specifications desired for selecting a seller.

The submitted task file is parsed and a thread is spawned for each sub-task. The thread created is responsible for looking up an appropriate seller node, negotiating the price for sub-task execution, and finally obtaining the results of computation and storing them in the output file(s) specified by the user.

User is also informed about the status of jobs running on his machine and status of any jobs submitted by the user for remote execution.

4.3.4 File Storage

The "File Storage" tab allows a user to back up its local data on multiple remote machines in the network. As part of our future work more functionalities will be added to this tab.
4.4 Remote Execution API

In addition to the GUI-based interface, one can use our RemoteExecution API for submitting a task file to the CompuP2P system. We have provided a TCP/IP socket interface for allowing the RemoteExecution API, which is in Java, to be usable by applications written in other languages. Applications supply the task file over the socket connection, and are provided a notification (of success or failure) when all the sub-tasks defined in the task file are finished executing.

4.5 Bootstrap Server

A node first enters the network by contacting a bootstrap server running at a well-known IP address and port number. This bootstrap server is referred to as AdminServer in our implementation. AdminServer, as shown in Figure 4.4, has information about all live nodes in the network, and returns the IP address and port number of a randomly selected existing node when contacted by a new node. The new node then uses this returned value to join the Chord network and update its routing table. Besides maintaining information about the live nodes in the network, the AdminServer also maintains information about the tasks being run by different nodes in the network. It is also possible to kill a node in the system, due to administrative policies or with an intent of stopping the system, from the AdminServer.

In our current implementation, we use AdminServer as a trusted bank that maintains an account for each node in the system. A node when it first enters the network is assigned some minimum currency that is credited to its account. Users’ accounts are automatically debited (credited) by AdminServer whenever they buy (sell) compute power as per the pricing strategy outlined in Section 3.2. Buyers with insufficient balance are not permitted to use computing power of others in the network.
4.6 CompuP2P execution on multiple nodes

Consider an application when a user wants to share the computing resources of multiple machines that he owns. A concrete example can be a system administrator of a computing lab, who wants to share computing resources of all the machines in the lab. This would require him to start the CompuP2P application on each of these machines, specify the constraints on the computing resource usage and share the computing resources.

To simplify this task we wrote a shell script which allows a user to make a remote login in multiple machines using SSH [24] [25] in parallel and start the CompuP2P application. We made changes in the CompuP2P implementation so that a node can join the CompuP2P network and specify the constraints in a file (formatted in XML), shown in Figure 4.5, instead of specifying them through GUI interface. Thus the process of joining the CompuP2P network, specifying constraints and sharing the computing resources is automated by the shell script and can be done from one computer. Similarly, we have written a shell script which allows a user to terminate the CompuP2P application.
<?xml version="1.0" ?>
<!- This file is used for text version of CompuP2P -->
<!- Specify Resource Constraints -->
<ResourceConstraints>
<MinLoadAllowed>80</MinLoadAllowed>
<MaxLoadAllowed>99</MaxLoadAllowed>
<!- Specify Max Time for the task to run. (can be empty) -->
<TimeForTask></TimeForTask>
<!- Specify Time of days when CPU cycles will NOT be shared -->
<!- By default it is shared at all times--> 
<!- Values- Morning, Afternoon, Evening, Night--> 
<NotSharedTimes>
<!- example <TimeOfDay>Night</TimeOfDay> --> 
</NotSharedTimes>
<!- Block the following IP addresses--> 
<BlockIP>
<Count>0</Count>
<IPAddress></IPAddress>
</BlockIP>
<!- Cpu cycles are shared or not- yes or no--> 
<CPUCyclesShared>yes</CPUCyclesShared>
</ResourceConstraints>

Figure 4.5 User constraints specified in XML file.

remotely from the multiple machines where CompuP2P was started.
CHAPTER 5. Fault Tolerance Computing

Fault tolerance is the property of a system that continues operating consistent with its specifications even in the event of failure of some of its parts. From a user's point of view, a distributed application should continue despite failures. In this chapter, we explain the various fault tolerance schemes implemented in CompuP2P which ensures the availability of computing resources in a distributed environment where the nodes can leave and join arbitrarily.

5.1 Serverless Checkpointing

It is possible that a computing node is not able to finish the computation assigned to it, either because it leaves the network, crashes, or the computation takes longer to complete than initially anticipated by a client. Under such circumstances, it may be expensive to restart the computation all over again. To handle such cases, periodic checkpoint of the computing node's state is taken so that, if required, the failed computation can be migrated to another node in the network.

Unlike traditional checkpointing, which relies on dedicated checkpoint servers to store the processing state, we propose to use server-less checkpointing in which nodes that store the checkpoint data are determined dynamically. Similar to the technique outlined in Section 4.2 for the sharing of compute power, we construct markets for memory storage. The client based on its estimation of the amount of checkpoint data can reserve the required memory space. The nodes performing computation are made
aware of such nodes, to which they periodically send a checkpoint of their computations. Upon failure of a computing node, the stored checkpoint data can be used to re-start the computation at other suitable node in the network.

It is possible that a user asks for checkpointing (by selecting the option in task file) but there is no node in the network sharing its memory storage. In this case, user is notified that checkpointing of intermediate results will not take place and has following options:

- User may execute the task on an appropriate seller of compute power without checkpoint data being stored on a separate node in the network.
- User may stop CompuP2P executing the task and try later after some time when there is availability of nodes in the system sharing their storage.

The checkpointing protocol built into CompuP2P is illustrated by Figure 5.1.

![Figure 5.1 Serverless Checkpointing in CompuP2P.](image)

In the figure, the service layer at a user node selects a computing node to which it sends the task for execution, and at the same time selects a storage node that has
sufficient disk capacity to store the periodic checkpoints generated by the task. Figure 5.2 describes the steps followed by CompuP2P to implement serverless checkpointing. The storage node is the checkpoint server for the task in consideration, and is selected based on the size of the checkpoint data produced by the task. The size value is specified by the user and is equal to the total size of all the objects that are needed to re-start the task. Our server-less checkpointing protocol is designed to take into account the failure of both the computing as well as the storage nodes. Failure of the storage server is monitored by the computing node, where as failure of computing node in turn is monitored by the service layer of the user.

/* U is the user's service layer, C is the computing node, and S is the storage node */

Activity 1:
U monitors C
if C fails
then
   Step 1: U finds another suitable computing node C'
   Step 2: Instructs C' to retrieve the last stored checkpoint data from S

Activity 2:
C monitors S
if S fails
then
   Step 1: C finds another suitable storage node S'
   Step 2: C notifies U, and gives the information about S'

Figure 5.2 Anonymous lookup protocol steps

Further, in practice errors in computation and/or communication of results can occur. Computation errors can occur due to faulty software/hardware at the computing node, or when a malicious node deliberately produces incorrect output. Such errors might be
hard to detect and correct. To increase the correctness of the end results the following alternatives may be used:

- Redundant computations, as also used in SETI@Home [3], may be employed. Basically this scheme involves performing the same computation multiple times at different nodes and then selecting the result produced by the maximum number of computing nodes. This capability is available to the user through the "Task Submission" tab in our CompuP2P GUI. Also, the user is notified about the discrepancies in the results obtained along with the IP addresses of nodes that generated those results. Note that redundant computations, like server-less checkpointing, also serve to address the problem of computing nodes failing or leaving the network. However, for long-running tasks checkpointing provides a better guarantee on the timeliness of the final result(s). We believe that best performance is achieved (although at a higher cost) by combining both these fault-tolerance strategies.

- The tasks may be designed in a way that certain characteristics of the answer are known in advance to the client, but hard to deduce just from the task code. In these cases, an answer that has these characteristics may be assumed correct.

- Some tasks may return answers that are easily verified correct. For example, a task, which solves an equation using some complex method, may be easily verified by plugging the solution into the equation also called acceptance test.

The above methods are available to users in CompuP2P and according to needs may be employed. However, all the fault-tolerance features come at an increased cost to a user and must be part of computing budget.
5.2 Stabilization of CompuP2P

CompuP2P is a dynamic system and the nodes can join and leave voluntarily. In Section 2.1 and Chapter 3, we discussed the protocols for:

• how new nodes join the system,

• how computing resources are shared,

• how the resource markets are constructed in CompuP2P.

In this section we discuss and analyze how the system stabilizes in the event of nodes joining or leaving the system. We enlist our goals as follows:

• Reachability of existing nodes is maintained.

• Failure of MO node should not affect the availability of the sellers listed in the markets maintained by the node.

• If the node leaving the network is seller then the concerned MO should remove it from the market.

5.2.1 Reachability

Nodes reachability should be maintained even in the event of concurrent node joins or simultaneous node failures. We have implemented methods described in Chord [4] to handle both events.

To handle concurrent node joins, each node in the system runs a Chord stabilize subroutine periodically. This subroutine corrects the routing table. More importantly each node corrects pointers to its immediate predecessor and successor. To handle simultaneous node failures, every node in the system, arranged in Chord identifier circle, maintains IP addresses and identities list of its \( k \) nearest successors on the Chord identifier ring. Any new node \( n \) learns of its successor when it first joins the Chord ring, by
asking an existing node to perform a lookup for n's successor; n then asks that successor for its successor list. The $k$ entries in the list provides fault-tolerance - if a node's immediate successor does not respond, the node can use the next live node's entry in its successor list. All $k$ successors would have to simultaneously fail in order to disrupt the Chord ring, an event that can be made very improbable with modest values of $k$.

The following results from [4] prove that reachability of nodes and correctness of Chord lookups is maintained in the events of concurrent joins or simultaneous node failures.

**Theorem 1:** If we take a stable network with $N$ nodes, and another set of up to $N$ nodes joins the network with no routing table pointers (but with correct successor pointer), the lookups will still take $O(\log N)$ time with high probability.

**Theorem 2:** If we use a successor list of length $k = O(\log N)$ in a network that is initially stable, and every node fails with probability $1/2$, then with high probability the search for successor return the closest living successor to the query key.

### 5.2.2 Node Failures

Failure of a $MO$ node can result in query failures for a seller, even though the seller is present in the network. In a loosely-organized network of peers seller listing needs to be replicated to avoid query failures. Thus we propose an extension to the method described in Section 4.2. A seller is listed at $m + 1$ nodes in the CompuP2P system. Thus for each seller there are $m + 1$ $MO$ nodes. A fixed value of $m$, as specified by the user, is used in CompuP2P.

In this new scheme, a seller first finds the successor node of hash($I$) on the Chord identifier circle, where $I$ is the value of seller's idle computing resource, as explained in Section 4.2. The successor node of hash($I$) becomes primary $MO$ for the seller and the seller forwards a request to enlist itself at the primary $MO$ node along with the value of $m$. The primary MO forwards this request to its successor node in the Chord identifier
circle but decreases the value of \( m \) by one. This process continues till the value of \( m \) becomes zero and it ensures that the seller is listed at \( m + 1 \) MO nodes in the system. Thus \( m \) nearest successors of the primary MO node on Chord identifier circle become the back-up MO nodes for a seller. Figure 5.3 shows the above process with the value of \( m = 1 \) and N1, N2 are nearest successors of MO node on the Chord identifier circle. Here N1 node assumes the role of back-up MO.

A query for a seller is made by finding the successor of hash\((R)\), where \( R \) is the required quantity of computing resource, as explained in Section 4.2. This query reaches the primary MO node which returns the appropriate seller. But if the primary MO node fails then the query reaches the first back-up node as it is the successor of primary MO on the Chord identifier circle. Thus the seller entry is returned ensuring the availability of seller in the system even when the primary MO node has failed. Note that in the worst case a MO node will have entries of all \( N \) sellers in the system. In Section 6.2, we estimate overhead incurred by a MO node due to the size of seller table.

A node leaving the network may be a seller which is listed on \( m + 1 \) MOs in the
network and this may result in incorrect entries at the MOs. In CompuP2P, each seller periodically tries to enlist itself by following the above protocol and each MO node, primary or back-up, maintains a timestamp corresponding to the seller entry indicating when the last enlist request was received. On receiving an enlist request for a seller which is already listed, the MO node simply updates the value of timestamp with the current system time. The period, say T, of sending an enlist request is average node failure rate in the system and is a global parameter in CompuP2P. A check is made at each MO node after time period T and all seller entries are removed for which enlist request has not been received for more than time 2T. This scheme is the similar to the health check scheme proposed in [18]. Lastly, before sending a reply to seller lookup MO checks if the seller is still alive and sharing the same idle capacity of the computing resource as listed.

In Figures 5.4 and 5.5, we show that periodic enlisting by sellers also ensures that a seller entry is always replicated at m nodes in the system. Figure 5.4 shows that in the event of primary MO node failure, the seller enlist request reaches the first back-up MO

Figure 5.4 Replication of seller entries when primary MO node fails.
node which forwards the request to its successor node on the Chord identifier circle. Similarly, in Figure 5.5 we show how the failure of a back-up node is handled. Note that since the nodes are arranged in an overlay network, the propagation delay shown between the nodes is an approximation.

A seller lookup issued after a node failure but before Chord stabilization subroutine, which corrects the routing tables, may fail as it may need to be routed through the failed node. Since we follow the Chord protocol for routing and the seller lookups, we state the following result from [4]: A failure rate of $k\%$ is observed if there are $k$ failures between stabilizations, assuming lookups are generated at a Poisson rate of one per second. To minimize seller lookup failures, in CompuP2P, we allow user to specify the number of attempts which the service layer makes before reporting a failure to the user. In Figures 5.4 and 5.5, we assume that stabilization has taken place after the node failures and this can be easily achieved by keeping stabilization period lesser than time period between enlist request.
5.2.3 Node Joins

Each seller in CompuP2P is listed on the successor node of the Chord identifier of seller's idle computing resource. A new node joining the network may become the successor of some seller's Chord identifier of idle computing resource, and this may lead to lookup failures for the seller. To avoid this seller lookup failure, each node joining the system after getting the successor's address on the Chord identifier ring, as described in Section 5.2.1, also asks its successor to transfer the relevant sellers to it. New joins may increase the number of nodes on which a seller is listed, but as explained above, if a node does not receive an enlist request for more than time period 2T for a seller then that seller entry is removed.

5.3 Experiments to test Fault Tolerance in CompuP2P

To demonstrate the fault tolerance property of CompuP2P, we injected faults in a stable system of 20 nodes. Specifically, we tried to measure the effect of MO node or seller node failures. In CompuP2P, when a computing node fails, an exception is raised and is detected by the Schedule Task subroutine. A query is then made to find another appropriate seller in the network after a time period which should be greater than the Chord Stabilize subroutine period, as this allows the system to stabilize before a new query is made. The number of attempts system makes after the detection of computing node failure is defined by the user.

The system settings are shown in Table 5.1, where T denotes the time after which each seller sends its enlist request to its primary MO node.

MO or seller node failure: We executed nQueens problem (refer Chapter 6), on three nodes in the system with the board size being 15 and the problem was equally distributed on three sellers of compute power in the system. We injected faults in the
Routine / Request | Time Period in seconds
---|---
Chord Stabilize | 8
T | 20
Schedule Task | 10

Table 5.1 Time Period for various routines in CompuP2P.

system by allowing one node failure in the system at random times when the task was being executed. We ensured that the nodes at which task is being executed do not fail. The average execution time for the problem after 20 runs was observed to be 21.608 seconds. This time is comparable to 21.55 seconds observed when we executed the same task without failures in CompuP2P, proving that underlying topology change in the overlay network does not effect the computation tasks.

**Serverless Checkpointing:** Server-less checkpointing is implemented as a part of service layer as shown in Figure 2.1. To checkpoint an application, a computing node needs to take a periodic checkpoint of the state of the application (the period is determined by the application). Additionally, before sending the task for remote execution, a client specifies in the task file that checkpointing is to be done and what is the size of the checkpoint data, as explained in 4.3.3.

To demonstrate checkpointing, we implemented an application of matrix addition in Java. In this application, addition of two large square matrices is done. In the checkpoint data we store the matrices to be added and the intermediate result. To take checkpoints, we used the Object Serialization feature provided in Java.

In this experiment we tried to observe the effect of the computing node failure at a random time once the task had started. Failure of the computing node leads to failure recovery mechanism in CompuP2P and a new computing node is found where the task starts from the last checkpoint state. After 20 runs the average time of execution was observed to be 26.017 seconds, while execution time without any failure is 15.246
seconds. Since the Schedule Task subroutine waits for 10 seconds before finding another computing node we notice the system latency in the results.

**Failure of Computing Node:** In this experiment we allow the computing node, executing a task, to fail at random times without checkpointing. We took the nQueens problem with a board size of 14 and the task was executed only on one node. After 20 runs the average time observed was 23.001 seconds, while the time to execute the task without failure was 8.215 seconds. Notice that since the task has to restart on computing node failure, the average time has to be greater than 18.215 seconds (Time to execute the task + Schedule Task latency).
CHAPTER 6. Applications and Overhead

We list here some applications of CompuP2P system. These applications are ones which can utilize the idle computing power in P2P systems. A common characteristic of all these applications is that they are all loosely coupled, i.e., they can be broken into rather independent sub-tasks, each heavy in terms of processing power requirements, but relatively light in terms of communication requirements. This characteristic is also called coarse grained parallelism. Some of such applications are brute force search, code breaking, simulated annealing, and Monte-Carlo simulations. All of these applications primarily involve generating many solutions in parallel and then using the solutions to come up with an answer for the initial problem. We use the following problems for demonstration.

**Traveling Salesman Problem:** In Traveling Salesman Problem (TSP), there are N cities and the distances from city i to city j is \( d_{ij} \). A tour is a path that starts from a city, visits each city exactly once, and goes back to the starting city. The goal, is to find a tour with minimum tour length. TSP is a well known NP-hard combinatorial optimization problem and Genetic Algorithm is one approach to solve it.

Genetic Algorithms (GAs) are adaptive heuristic search algorithm based on the evolutionary ideas of natural selection and genetics. As such they represent an intelligent exploitation of a random search used to solve optimization problems. Briefly, GAs simulate the survival of the fittest among individuals over consecutive generation for solving a problem. Each generation consists of a population of character strings that are analo-
gous to the chromosome that we see in our DNA. Each individual represents a point in a search space and a possible solution. The individuals in the population are then made to go through a process of evolution.

We use GAs to solve TSP for measuring the speed up on CompuP2P between the Sequential Genetic Algorithm (SGA) and Parallel Genetic Algorithm (PGA). We used the code from GATSS [27], and made some modifications in it for our experiments. A 52 city symmetric ($d_{ij} = d_{ji}$) Euclidean TSP instance was chosen as a test case from TSPLIB [28].

For our experiments, in SGA, we chose an initial population of 1000 tours (chromosomes) and the simulation stops when a better solution is not reached for 500 iterations (generations in GA terminology). To test SGA, a total of 20 runs were conducted.

Simplest way of parallelizing a GA is to execute multiple copies of the same SGA, each one on different processor. On CompuP2P, 5 processors (5 sellers) were chosen and on each processor, SGA starts with different initial subpopulation (1000/5 chromosomes), evolves and stops independently. The complete PGA halts when all processors stop. This approach has two advantages:

- it has coarse grained parallelism with no inter-communication between processors,
- subpopulation diversity reduces the chance that all processors prematurely converge to the same poor quality solution.

This approach is equivalent to simply taking the best solution after multiple executions of the SGA on different initial populations [29]. Table 6.1 gives the average values of the execution time for SGA and PGA in seconds. The speedup obtained by the execution of PGA on CompuP2P was 4.3 which is approximately equal to the number of processors used. Again this speedup was obtained by using idle computing resources over the internet.
### Table 6.1 Performance Comparison of SGA and PGA.

<table>
<thead>
<tr>
<th></th>
<th>SGA</th>
<th>PGA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average</td>
<td>28</td>
<td>6.5</td>
</tr>
</tbody>
</table>

**N-Queens Problem:** N-Queens problem involves finding number of ways in which N queens can be placed on a chessboard of size NxN such that no queen can take another. In the problem, a queen can take another piece that lies on the same row, the same column, or the same diagonal (either direction) as the queen. This problem can be solved with *backtracking*. Backtracking algorithms attempt to complete a search for a solution to a problem by constructing partial solutions, always ensuring that the partial solutions remain consistent with the requirements of the problem. The algorithm then attempts to extend a partial solution toward completion, but when an inconsistency with the requirements of the problem occurs the algorithms backtracks by removing the most recently constructed part of the solution and trying another possibility. A sequential backtracking algorithm for the N-Queens problem is shown in Figure 6.1. Initially, we start with a blank chessboard of size \( N \) and column \( 1 \).

```
numSolutions=0
Place(column, Board)
for each row on the Board
do Board[column] = row // try to place a queen in current row
if (safe to place the queen in current row and column)
then if (column=N)
then numSolutions=numSolutions+1
else Place (column+1,Board)
elself Board[column]=0 // unrecord that a queen was placed
```

Figure 6.1 Algorithm for solving N-Queens Problem

As shown in Figure 6.3, this algorithm takes considerable time for a chessboard of size \( N = 10 \) or more if solved it on one processor. However, this algorithm can be made parallel by dividing the work and running it on more than one processor as shown in
Figure 6.2 Algorithm for solving N-Queens Problem on multiple processors

For example, if we have 2 processors and a chessboard of size = 8, first and second processor will run the code in Figure 6.1 and Figure 6.2 with \( start = 1 \) and \( end = 4 \), and with \( start = 5 \) and \( end = 8 \), respectively. Final solution is the sum of solutions computed by both nodes.

Figure 6.3 Comparison of parallel and sequential execution of N-Queens problem.

Figure 6.3 shows comparison of parallel and sequential execution of N-Queens problem. We plotted time (in milliseconds) on the log scale as the function of board size because the time to solve the problem increases exponentially as the board size increases.
For parallel execution using CompuP2P, with a board size of $N$, the work load was equally distributed on $N$ processors. We can observe that the time required to solve the problem for board of any size is significantly reduced. This speedup was possible simply by utilizing idle capacity of machines in the network.

6.1 PlanetLab Testbed and Experimentation

PlanetLab [22] is an overlay network which consists of nodes present in the Internet and provides a test-bed to researchers for conducting experiments, simulations etc. on a large set of geographically distributed machines. PlanetLab nodes can also be used for deploying and testing an application in a distributed environment.

We installed CompuP2P on 12 PlanetLab nodes and made them part of the CompuP2P system. We found our system extremely stable and we used it to solve nQueens problem. We observed that the nodes in PlanetLab are heavily utilized because many people around the globe might be running their tasks on them. Since CompuP2P aims at sharing of idle computing resources by the nodes present in the system, it was difficult to find sellers among PlanetLab nodes with sufficient compute power.

For example, Table 6.2 shows the comparison of time taken for sequential and parallel execution of nQueens problem. We used the backtracking algorithm described in Section 6. In Table 6.2, for solving board with size 15 and 17 three sellers of computing power were used, while four sellers were used for board with size 16. Problem was equally distributed among the nodes except for board of size 17; one node solved the board for 5 columns while the other two solved for 6 columns each.

6.2 CompuP2P Overhead

CompuP2P is a light-weight architecture and incurs minimal overhead on the system. It is built on top of Chord, which is a scalable, efficient, and robust protocol [4]. In this
<table>
<thead>
<tr>
<th>Board Size</th>
<th>Sequential Execution</th>
<th>Parallel Execution on PlanetLab nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>67.693s</td>
<td>21.989s</td>
</tr>
<tr>
<td>16</td>
<td>485.196s</td>
<td>106.464s</td>
</tr>
<tr>
<td>17</td>
<td>3899.42s</td>
<td>2542.65s</td>
</tr>
</tbody>
</table>

Table 6.2 Execution of nQueens problem on PlanetLab nodes

section we examine in detail the additional overhead incurred by the sellers, buyers, and MOs by the CompuP2P architecture. The overhead is in the form of either message communication or state maintained by each of these entities.

**State maintenance:** A buyer (seller) maintains the only the IP address of a seller (buyer). The information maintained by MOs is also minimal. This information size, $S$, is given by

$$S = n \times a \times s$$

(6.1)

where $n$ is the number of sellers in a market, $a$ is the number of different attributes of a seller, and $s$ is the space required to store a value of each attribute. To see how much the value of $S$ evaluates to, let us consider a MO that stores information about 10,000 sellers. The MO might store several attributes pertaining to a seller. These attributes include information regarding a seller’s IP address, marginal cost, OS type, OS version, processor configuration, etc. Suppose that there are 10 attributes value for each seller, and each attribute require 4 bytes of memory space. Then, the total information maintained by the MO is equal to, $10,000 \times 10 \times 4 \approx 400KB$. Thus, even for a large-sized market, its state information ($< 0.5MB$) can easily reside in any modern PC’s RAM, which are typically 512MB. Furthermore, since the entire market information easily fits into a MO’s main memory, lookups to select an appropriate (based on a client’s request) seller are also very fast.
**Message communication:** In the Hybrid Scheme, implemented in CompuP2P to construct markets, messaging overhead incurred by both buyers and sellers to locate a market is $O(\log N)$ where $N$ is the number of nodes in the system. Once a buyer has selected a seller for service further communication between them takes place using a direct TCP/IP connection, bypassing Chord routing. Message communication overhead incurred by MOs is almost negligible (apart from direct TCP/IP connections with buyers and sellers).

For the replication of sellers on $m + 1$ MO's, the seller makes a Chord lookup before sending the enlist request after every $T$ time period, as mentioned in Section 5.2.2. Since each MO, primary or back-up, knows about its successor on the Chord identifier circle, the enlist request is forwarded by direct TCP/IP connection.
CHAPTER 7. Comparison with Related Projects and Conclusion

In this chapter we present a comparison of CompuP2P with current Distributed Computing projects, before concluding the thesis.

7.1 Related Distributed Computing Projects

CompuP2P is an architecture for enabling Internet computing, and is thus significantly different from large-scale distributed computing projects that have been implemented in the arena of grid or public resource sharing computing. These differences were highlighted in Chapter 1. Here we compare CompuP2P with some specific well-known projects, such as Condor [15], Entropia [11], SETI@home [3], and POPCORN [14], to bring out the novelty and usefulness of this new architecture.

Condor is designed to harness the idle CPU cycles of workstations, desktops, servers etc. Users submit their sets of serial or parallel tasks to Condor in form of jobs. The Condor matchmaker decides where to run them based on job needs, machine capabilities and usage policies. Task management is centralized to ensure that jobs are executed based on the specified requirements of provider and consumer.

Entropia is a commercial product, and is sold as part of Entropia’s DCGrid enterprise solution (www.entropia.com). Since the majority of desktops are Windows x86 machines, Entropia focuses purely on providing a Windows x86-based solution, supporting three generations of Windows operating systems NT, 2000, and XP. The Entropia system
architecture is composed of three separate layers. At the bottom is the Physical Node Management layer that provides basic communication and naming, security, resource management, and application control. On top of this layer is the Resource Scheduling layer that provides resource matching, scheduling, and fault-tolerance. Users can interact directly with the Resource Scheduling layer through the available APIs or alternatively, users can access the system through the Job Management layer that provides management facilities for handling large number of computations and files.

Unlike Condor and Entropia, CompuP2P is completely decentralized, in the sense that there is no centralized entity that monitors system state and assign (sub)tasks accordingly. CompuP2P use microeconomic principles and game-theoretic ideas to govern trading and allocation of compute power to tasks. Moreover, CompuP2P is implemented using Java and can theoretically run on virtually any system. Entropia on the other hand is designed for Windows-based system only.

In SETI@home, only one central node can allocate tasks to others, whereas in CompuP2P all the grid nodes can purchase compute power and distribute their workload to other machines.

POPCORN provides an infrastructure for globally distributed computation over the whole Internet and uses a market-based mechanism to trade CPU cycles. However, unlike in CompuP2P, POPCORN uses a trusted centralized market that serves as a matchmaker between the seller and buyer nodes.

Sharing of CPU cycles in CompuP2P is completely distributed and fault-tolerant as compared to the scheme proposed in [16] that uses a centralized auction.

7.2 Conclusion and Future Work

In this thesis we have discussed CompuP2P, which enables Internet computing. CompuP2P is significantly different from other large-scale distributed computing projects
which have been implemented in the arena of grid or public resource sharing computing. It can be used for building large Internet computing infrastructures, and can potentially reduce the need for expensive processing or storage servers in an enterprise, for example. Users of CompuP2P can harness almost unlimited processing capacity of the entire network in a complete distributed manner without using any centralized administrative authority.

The future work complimenting the works presented in this thesis may include following:

- Design and implementation of a workflow engine [19] and integrating it with CompuP2P. Workflows allow dependencies between sub-tasks to be represented in the form of an acyclic graph, and are an important business tool.

- Our current implementation does not allow subtasks to communicate with each other. Additions can be made in CompuP2P code which will allow subtasks to share objects between them.

- Implementation of a completely distributed virtual currency system, like KARMA [17].

- As mentioned in Section 4.3.4 and 4.3.2, resource sharing and file storage tab in the current implementation of CompuP2P may be extended with more functionalities.

- Implementation of a Code Server. Currently, the executable file for a task is assumed to be locally available or searched for in the network like any other shared file. To ensure more security a well known code server can be included in the system, which can provide a digitally signed copy of the code.
APPENDIX

In this Appendix we will give a brief introduction on game theory and the concepts used in this thesis. For more details the reader is referred to [8]

Game Theory

Game theory is a formal study of decision-making where several players must make choices that potentially affect the interest of the other player. Here players may be individuals, groups, firms or any combination of these.

Nash Equilibrium

In strategic game, Nash equilibrium is a list of strategies, one for each player, which has a property that no player can unilaterally change his strategy and get a better payoff.

Prisoner’s Dilemma

In Prisoner’s dilemma problem, two burglars are being interrogated for a suspected crime. They can not communicate between each other. Each has to choose whether or not to confess and implicate the other. If both confess the crime and implicate the other for the crime, then both of them get a term of 5 years. If both do not confess, they get a term of 1 year. But if one burglar confesses and implicate the other, then the other gets a term of 20 years and the first one goes out free. The situation is summarized in the Table A.1, where rows are for the first burglar and columns are for the second burglar.
Table A.1  Prisoner’s Dilemma

<table>
<thead>
<tr>
<th></th>
<th>Confess</th>
<th>Don’t Confess</th>
</tr>
</thead>
<tbody>
<tr>
<td>Confess</td>
<td>5,5</td>
<td>0,20</td>
</tr>
<tr>
<td>Don’t Confess</td>
<td>20,0</td>
<td>1,1</td>
</tr>
</tbody>
</table>

The Nash equilibrium, in this game, is both prisoners confess the crime and get a term of 5 years each. While best strategy for them would have been to not confess and get a term of 1 year each.

Noncooperative Oligopoly Models

Noncooperative oligopoly is a form of market where small number of firms act independently but are aware of each other’s actions. Some typical assumptions in oligopoly are consumers are price takers, there is no entry in the industry and all firms produce homogeneous products. We will discuss two noncooperative models here.

In Cournot’s oligopoly model, firms are faced with the problem of how much to produce and then let the demand conditions set the price of the product. As an example, suppose we have a market with two firms A and B, and with inverse demand curve \( P = C - Q \), where \( P \) is the price, \( Q \) is the total quantity in the market, and \( C \) is some positive constant. Then both firms have to decide how much should they produce to maximize the profit.

In Bertrand’s oligopoly model, firms are faced with the problem of determining the price of their product which determines the demand for the product. In this model, assuming that each firm has same marginal cost, it turns out that the firms can not price their product above their marginal cost.
BIBLIOGRAPHY


[3] SETI@home, project to search for Extraterrestrial Intelligence using Internet-connected computer (Documentation). http://setiathome.berkeley.edu/sah_about.php (Date retrieved: 19th January, 2005).


[22] PlanetLab, a test-bed to perform experiments (Documentation). http://www.planet-lab.org/ (Date retrieved: 13th July, 2005).


ACKNOWLEDGEMENTS

I would like to take this opportunity to express my thanks to those who helped me with various aspects of conducting research and the writing of this thesis. First and foremost, Dr. Arun Somani for his guidance, patience and support throughout this research and the writing of this thesis. But for his support, I would have never been able to complete this work. People in DCNL were always close, friendly and helpful. Especially, I will thank Rohit Gupta whose feedback and comments were helpful throughout this work.

This research was supported by Nicholas Professorship funds and Jerry R. Junkins Endowed Chair Professorship funds. I would like to thank Iowa State University for the support.

I express my deepest gratitude to my parents for giving me the support throughout my education. I felt their blessing, support and encouragement always with me.

Finally, my thanks goes to all who have contributed directly or indirectly in successful completion of this work.