CompuP2P: A Light-Weight Architecture for Internet Computing

Varun Sekhri, Rohit Gupta, and Arun K. Somani
Dependable Computing and Networking Laboratory
Department of Electrical and Computer Engineering
Iowa State University, Ames, IA 50011
E-mail: {varuns, rohit, arun}@iastate.edu

***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 share their computing resources, and computations 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. CompuP2P is a light weight architecture for enabling Internet computing. It uses peer-to-peer networks for sharing of computing resources. CompuP2P create 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.**

This paper discusses the system architecture, functionality, and applications of the proposed CompuP2P architecture. We have implemented a Java based prototype, and 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.

I. 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 for CompuP2P can be- a) allowing processing limited device, such as wireless clients, to distribute their processing requirements to other machines in the network, and b) utilizing the storage capacity of virtually millions of machines connected to Internet, etc.

Peer-to-peer (P2P) networks [1], [2], [3], [5], [6] are flexible distributed systems that allow nodes (also called peers) to act as both clients and servers and provide services to each other. 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.

This paper discusses the system architecture, functionality and applications of the proposed CompuP2P architecture. CompuP2P uses peer-to-peer networks for sharing of computing resources. It create 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 use ideas from game theory [9] and microeconomics [11] to devise incentive-based schemes for motivating users to share their computing resources with each other.

CompuP2P is designed to be light-weight in terms of protocol overhead and simplicity in control mechanisms. It uses P2P substrate, a Chord [5] overlay network, to provide lower level services for example connecting peers. Above this substrate different light weight control mechanisms have been built to create markets dynamically. Search for appropriate markets, trading and pricing of resources is done in a completely distributed manner without requiring any trusted and/or centralized authority to oversee the transactions. Number of messages exchanged to look up an appropriate market is $O(\log N)$, where N is the number of nodes in a network.

We have implemented a Java based prototype of CompuP2P architecture, and 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.

The rest of the paper is organized as follows. Section 2 compares Internet computing with grid computing and public resource computing. Section 3 gives an overview of CompuP2P system architecture. Section 4 describe the protocols for market creation and resource pricing. In Section 5, we present our prototype implementation of CompuP2P, while in Section 6 we draw differences between CompuP2P and other existing large scale distributed computing projects. We conclude the paper in section 7.

0-7803-9277-9/05/$20.00/©2005 IEEE

II. Grid Computing 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 [8] 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 [7] 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 heterogeneous 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 control 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 present several unique and interesting research challenges. These issues, along with how they are addressed by the CompuP2P architecture, are discussed throughout the remainder of this paper.

III. CompuP2P: System 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 things such as 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 markets 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 use ideas from game theory and microeconomics for pricing of computing resources. CompuP2P allow users to define their policies regarding what, when, how, and by whom their resources can be used. Moreover, it allow 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.

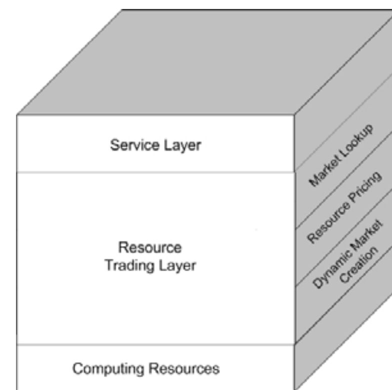


Fig. 1. CompuP2P: system architecture

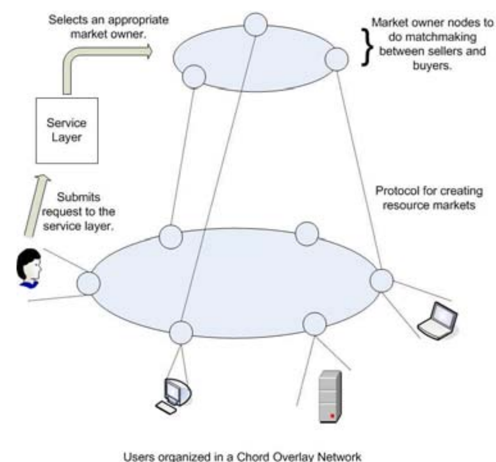


Fig. 2. A conceptual view of CompuP2P

Figure 1 depict the layers constituting the CompuP2P architecture and are explained in the following subsections. Figure 2 depicts a conceptual view of the operation of a CompuP2P system.

A. Computing Resource layer

This layer refer 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 joining the system are organized in a Chord ring [5]. 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. Briefly, 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 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. 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 [6].

B. Resource Trading layer

As shown in Figure 1, the functionality of this layer can be further divided into three sub-layers:

- *Market lookup protocol*: It ensure that sellers and buyers looking to trade a commodity converge on the same market.
- *Resource pricing protocol*: The pricing mechanism ensure 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.

C. Service layer

The service layer accept 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 an XML 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.

D. CompuP2P Usage Steps

To demonstrate the working of the overall system, we briefly describe the steps taken by a user to carry out a distributed computation.

1) The user submits an XML formatted 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 appropriated 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 MO node is queried about the available sellers. The MO acts as a match-maker 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.

IV. 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 we use compute power 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.

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 we would use another appropriate unit like G gigabytes for T time units. It must be noted that the same value of 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. We use the unit of cycles/sec 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. We address these issues in the following sections.

A. 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 scalability, fault-tolerance, and security problems.

For efficient creation and lookup of compute power markets, we propose two schemes 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 depicts how nodes with different values of idle compute power *C* join different markets. Although, for simplicity of discussion, we have used *C* as a discrete value, 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.

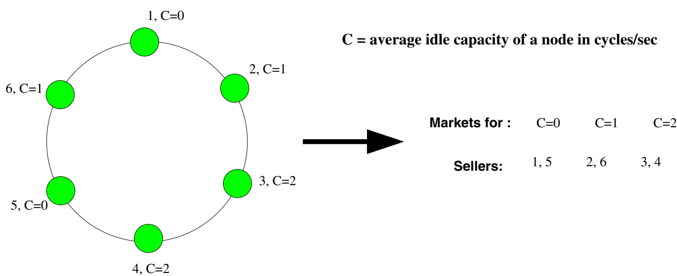


Fig. 3. Creation of markets for CPU cycles in CompuP2P.

We describe below two schemes for the creation of compute power markets.

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.

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 *MO*s, 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. We assume that the value of *c* is 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 4. 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 (= $hash(C)$, a value between 0 and $2^m - 1$. The successor node of $hash(C)$ is then the *MO* for the market trading in commodity *C*. The various *MO*s 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 *CMID*s 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 4.

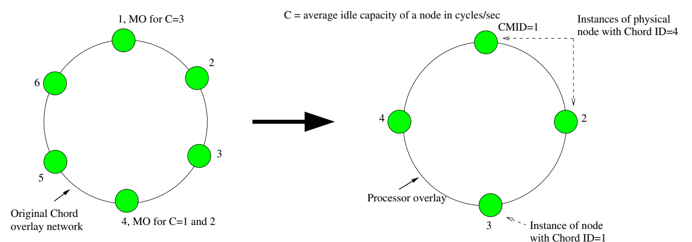


Fig. 4. *Processor overlay* schema using the CPU capacity values given in Figure 3.

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 *MO*s 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.

B. Resource Pricing

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 modelled as a one-shot game and using the results from game theory and microeconomics (the classical Prisoner's dilemma problem [9] and Bertrand oligopoly [11], respectively), we can see 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 [9] for a game-theoretic derivation of this result). In Bertrand oligopoloy 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*.

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.

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 we describe two such strategies depending on whether fixed or variable listing pricing is used to compensate a *MO*.

1) 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 [10]. 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 [10] 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 $(MC^{-1} - MC)$.

2) 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 than 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 $1, 2, \ldots, S$, 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.

$$\text{Payoff}_{\text{MO}} = (MC'_S - MC'_1)/(MC'_S)^2 + \delta$$
$$\text{Payoff}_{\text{seller}} = MC'_1 + 1 \qquad (1)$$

MC'_1 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, δ 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. We formalize this in the form of the following lemma.

Lemma 1: Assuming one-shot model of compute power transactions, the payoff strategy in Equation 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.

Proof: a) The *MO* can increase its payoff by reporting a low value for the lowest listed marginal cost, i.e., minimizing MC'_1 as much as possible. However, MC'_1 cannot be decreased below MC_1, the true lowest marginal cost, since otherwise the seller (here seller *1*) gets a payoff of $MC'_1 + 1 (\leq MC_1)$. Since a seller does not provide its service unless its payoff is greater than its marginal cost, the best strategy for the *MO* is to set $MC'_1 = MC_1$ and return the lowest cost supplier for processing.

b) This is implied from Equation 1 where we see that the payoff received by the seller is one more than its marginal cost.

c) From Equation 1, the payoff to the *MO* is maximized for $MC'_S = 2 * MC_1$ (after setting $\frac{\partial \text{Payoff}_{\text{MO}}}{\partial \text{MC}_\text{S}} = 0$), giving it a payoff of $1/(4 * MC_1)$. Note that in the given network model, it is difficult for a buyer to verify the marginal cost values it receives from the *MO*. Thus, the total cost to the buyer is bounded, and is equal to $1/(4 * MC_1) + \delta + MC_1 + 1$.

d) It follows from the description of the payoff values given by Equation 1.

■

In the above we assume 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.

V. PROTOTYPE IMPLEMENTATION OF COMPUP2P

We have implemented a Java-based prototype of the proposed CompuP2P architecture for the sharing of compute power, and have deployed it in our lab for running compute intensive simulations. Java owing to its platform independence and write-once run-anywhere feature enables easy migration of tasks from one node to another in a heterogeneous system.

Fig. 5. Screen snapshot of Usage Policy tab.

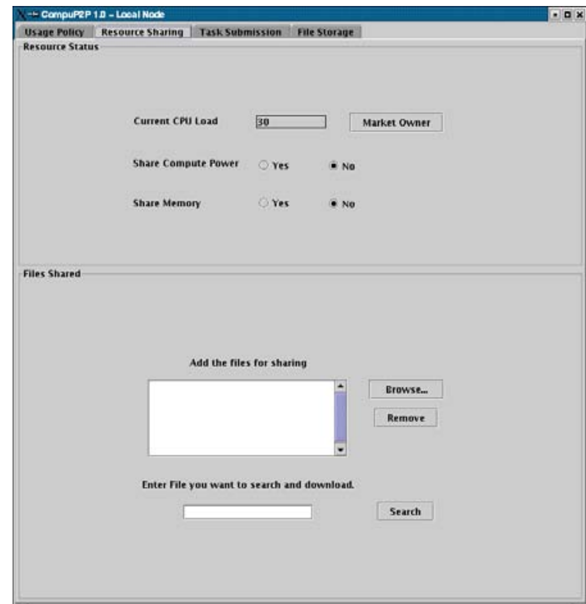


Fig. 6. Screen snapshot of Resource Trading tab.

Screen-snapshots of the implemented CompuP2P prototype, as it appears to a user, are shown in Figures 5 and 6. The

first tab, “Usage Policy”, allows a user to specify the usage constraints on the local shared resources. For example, a user can specify the CPU load levels (in percentage) beyond which the node is not allowed to share its compute power. To prevent a task from running forever, user can also impose the maximum allowable run time on tasks received for execution. User can specify specific times of the day 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. Moreover, for sharing storage space, user can specify the total allocated shared space, along with the directory name where the received files are to be stored. Furthermore, user can limit access to the machine by specifying the IP addresses of nodes that are not permitted to utilize the shared resources.

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 files, which can be downloaded by others in the network.

The “File Storage” tab allows a user to back up its local data on multiple remote machines in the network.

```
<?xml version="1.0" ?>
<TaskFile>

 <SubTask>
      <CpuCycles>1000</CpuCycles>
      <CheckPointDataSize>0</CheckPointDataSize>
      <Price>234</Price>
      <File>job.exe</File>
      <InputParameters>
               <ICount>2</ICount>
               <Param>3</Param>
               <Param>4</Param>
      </InputParameters>
      <OutputFiles>
               <OCount>1</OCount>
               <OutputFile>joboutput.out</OutputFile>
      </OutputFiles>
 </SubTask>

</TaskFile>
```

Fig. 7. XML-based task file

A user submits its task to the system in the form of a *task* file. 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 7. For each sub-task, the following information is included.

- *Code ID* (or name) of the executable file for the sub-task. The executable file (if not locally available) can be downloaded either from a well-defined code server or can be searched for and downloaded just as other normal data using code ID (or name) as the key. To ensure security, the computing node executes the downloaded code in an appropriate sand-boxing environment. 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.

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. The task file is supplied to the service layer via the “Task Submission” tab.

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 has information about all live nodes in the network, and returns the (IP address, 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.

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 IV-B. Buyers with insufficient balance are not permitted to use computing power of others in the network.

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 name 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.

The system is clearly intended for very coarse-grained parallelism. The efficiency is mostly determined by the ratio between the computation time of sub-tasks to the communication effort needed to send them and handle the overhead. To achieve high efficiency, sub-tasks should be relatively heavy in terms of computation time.

In order to address the nodes’ heterogeneity problem there are at least couple of possible alternatives. One alternative is to use the *SPECjvm98* benchmark [13]. to normalize the compute power values so that a given value is interpreted similarly by all the different nodes. Another alternative that is currently used in CompuP2P is to use a *MO* as a *matchmaker*. A *MO* now 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 meet the platform specifications as desired by a client for its sub-tasks.

A. Handling Failures of MOs and Listed Sellers

It is possible that nodes selected as *MO*s as well as sellers listed in those markets might fail. To account for such possibilities, we incorporate the following additional strategies in our prototype implementation.

1) Sellers periodically re-list themselves in a (new) market. This is done irrespective of whether the amount of a computing resource they are offering has changed or not. This periodic listing takes care of the following two problems that may arise in any dynamic system - a) *MO*s leaving the network, and b) new nodes joining the network that may replace some existing *MO*s as the new *MO*s for the respective markets.
2) Likewise, every *MO* periodically purges the information it maintain about the listed sellers. Thus, seller information is maintained as a soft-state information, and is never outdated for too long. Additionally, a *MO* before returning information about the selected seller to the buyer checks whether the seller is still alive (i.e., is part of the network) or not.

B. CompuP2P Applications

We list here several applications of CompuP2P system. 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. Some of such application 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 have found CompuP2P to be very useful for running large simulations. We used CompuP2P for running parallel simulations on multiple optical network topologies as generated by a user using ISTOS [14], which is an advanced tool for simulating fiber optic networks. Users in ISTOS can create multiple network topologies on which simulations are to be carried out. Earlier all simulations were sequentially executed on a single back-end server. However, now we can exploit the CompuP2P architecture to distribute the task of simulating different network topologies to different nodes in the network which agree to share their idle computing power. The resulting speedup was very close to the number of different nodes which were used in parallel to run the simulations. This high speedup was possible mainly because of the low communication overhead. (An executable file of the simulation code was pre-installed on the nodes).

Figure 8 shows the speedup achieved as a result of using the *RemoteExecution* API. In our experiments the number of computing nodes were 10, and the number of sub-tasks were successively increased from 1 to 10. It can be observed that CompuP2P provides substantial tremendous gain and this is achieved by simply utilizing the idle capacity of machines in the network. The speedup would become close to 1 as the task computation time is increased. Ups and downs seen in Figure 8 for the parallel execution time are due to the heterogeneity of processors used in the experiment.

C. Fault-Tolerant Computing

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, it is useful to periodically checkpoint the computing node's state, 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 on-the-fly. Similar to the techniques outlined in Section IV-A for the sharing of compute power, we can 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 another suitable node in the network. We use the object serialization feature provided in Java, which enables a failed computation to be continued at a different node upon failure of an initially allocated processing node. The checkpointing protocol built into CompuP2P is illustrated by Figure 9.

In Figure 9, 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. 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

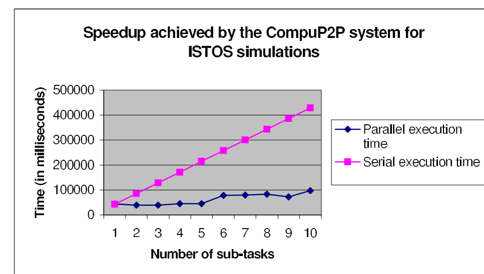


Fig. 8. Figure showing the total time required to execute sub-tasks using CompuP2P, as opposed to the time required when all the computations are carried out on a single node.

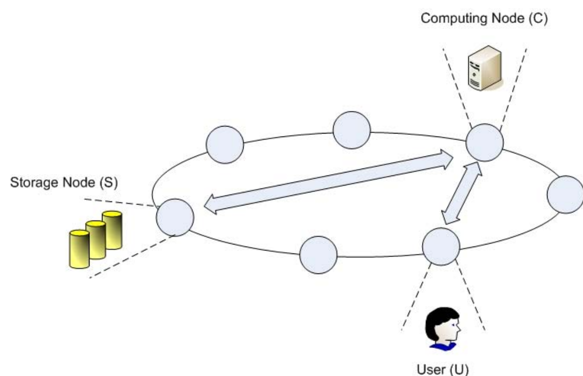
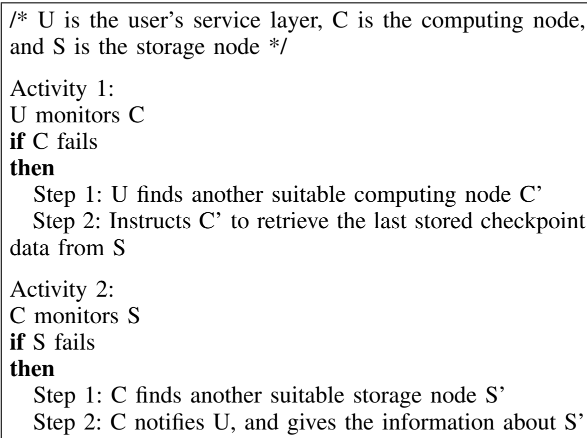


Fig. 9. Serverless Checkpointing in CompuP2P.

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. Figure 10 describes the steps followed, in CompuP2P, to implement serverless checkpointing.


/* 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'

Fig. 10. 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 reliability in the correctness of the end results the following alternatives can be used:

- Redundant computations, as also used in SETI@Home [3], can 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.
- 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.

However, all the fault-tolerance features come at an increased cost to a user. The user's budget should be sufficient to cover the cost of reserving memory space to store the checkpoint data and/or compensate the redundant computing nodes for their processing.

D. 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 [5]. In this section we examine in detail the additional overhead incurred by the sellers, buyers, and *MO*s by the CompuP2P architecture. The overhead is in the form of either message communication or state maintained by each of these entities.

a) Message communication: As discussed in Section IV-A, messaging overhead incurred by both buyers and sellers to locate a market is $O(\log N)$ in case of single overlay scheme, and $O(\log M)$ (where M is the number of different markets) for processor overlay scheme. 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 *MO*s is almost negligible (apart from direct TCP/IP connections with buyers and sellers).

b) State maintenance: In processor overlay scheme all nodes have to maintain additional (apart from the initial Chord-based overlay state) $O(\log M)$ routing information for maintaining the processor overlay, and in single overlay scheme no additional routing information is required.

A buyer (seller) maintains the IP address of a seller (buyer). Moreover, information maintained by *MO*s is minimal. This information size, S, is given by

$$S = n * a * s \tag{2}$$

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, $10,000 * 10 * 4 \approx 400\text{KB}$. Thus, even for a large-sized market, its state information ($< 0.5\text{MB}$) 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.

CompuP2P relies on a monetary payment scheme to compensate nodes for their computing resources. While the use of a monetary scheme provides a clean economic model, implementing the associated electronic payment infrastructure can be very expensive. In order to overcome this problem, in [18] we have proposed a framework for implementing a system of virtual currency by using reputation as a measure of nodes' wealth.

VI. 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 [8], [4] or public resource sharing computing [7]. Here we compare CompuP2P with some specific well-known projects, such as Condor [16], Entropia [12], SETI@home [3], and POPCORN [15], 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. 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 [17] that uses a centralized auction.

VII. Conclusion

In this paper 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.

As part of our future work, we would design and implement a workflow engine [19] and integrate 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.

References

[1] Kaaza. http://www.kaaza.com.

[2] Gnutella. http://gnutella.wego.com/.

[3] SETI@home. http://setiathome.ssl.berkeley.edu.

[4] http://www-unix.globus.org/toolkit/.

[5] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Protocol for Internet Applications. *In Proceedings of the 2001 ACM SIGCOMM Conference*, 2001.

[6] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. *In the Proceedings of ACM SIGCOMM (San Diego, 2001)*, 2001.

[7] D. P. Anderson. Boinc: A System for Public Resource Computing and Storage, *5th IEEE/ACM International Workshop on Grid Computing, Pittsburgh, USA*, November 8, 2004.

[8] I. Foster, and C. Kesselman. The Grid: Blueprint for a New Computing Infrastructure, 2nd Edition, Morgan Kaufmann, 2004.

[9] M. J. Osborne. A course in game theory. Cambridge, Mass. : MIT Press, c1994.

[10] W. Vickrey. Counterspeculation, auctions and competitive sealed tenders. *Journal of Finance, pages 8-37*, 1961.

[11] M. R. Baye. Managerial Economics and Business Strategy. *Third edition, McGraw Hill*, 2000.

[12] A. Chien, B. Calder, S. Elbert, and K. Bhatia. Entropia: architecture and performance of an enterprise desktop grid system. *Journal of Parallel and Distributed Computing, vol. 63, pp. 597-610*, 2003.

[13] Standard Performance Evaluation Corporation. *SPECjvm98 Documentation*, Release 1.0. August 1998. Online version at http://www.spec.org/osg/jvm98/jvm98/doc/index.html

[14] http://ecpe.ee.iastate.edu/dcnl/DCNLWEB/Tools/tools_ISTOS.htm.

[15] N. Nisan, S. London, O. Regev, and N. Camiel. Globally distributed computation over internet - The POPCORN project. *In the Proc. of 18th IEEE Int. Conf. Distributed Comput. Syst., pages 592-601*, May 1998.

[16] P. Wagstrom. An Overview of Condor. February 19, 2002.

[17] M. Senior, and R. Deters. Market Structures in Peer Computation Sharing. *Second International Conference on Peer-to-Peer Computing (P2P'02)*, 2002.

[18] R. Gupta, and A. K. Somani. A Framework for Reputation Management and Using Reputation as Currency in Large-Scale Peer-to-Peer Networks. *In the Proc. of Fourth IEEE International Conference on Peer-to-Peer Computing, ETH Zurich, Switzerland*, July 2004.

[19] Ian Foster, and Carl Kesselman. The Grid: Blueprint for a New Computing Infrastructure, 2nd Edition, Morgan Kaufmann, 2004.