CompuP2P: An Architecture for Sharing of Computing Resources In Peer-to-Peer Networks With Selfish Nodes

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: {rohit, arun}@iastate.edu

***Abstract*— CompuP2P is an architecture for sharing of computing resources in peer-to-peer (P2P) networks. It provide resources, such as processing power, memory storage etc., to user applications that might require them. CompuP2P creates dynamic markets for different amounts of computing resources without relying on any trusted centralized entity to monitor the activities of nodes in those markets. Moreover, the pricing of computing resources takes into account selfishness of network users and uses ideas from game theory and microeconomics.**

I. Introduction

CompuP2P is an architecture for sharing of computing resources in peer-to-peer (P2P) networks. It provide resources, such as processing power, memory storage etc., to user applications that might require them. For example, such a system can perform compute intensive tasks on behalf of clients, such as wireless devices (e.g. PDAs) with limited battery and processing power. Applications, like scientific simulations and data mining, requiring large processing requirements, can tremendously benefit from potentially unlimited availability of compute power provided by CompuP2P. Likewise, database applications, requiring huge storage, can harness the disk capacity of virtually millions of machines connected to the Internet.

At present, P2P networks, such as Napster [1], Gnutella [2] etc., are used primarily for ”data sharing”. Although, it is widely acknowledged that other resources, like compute power, can also be shared using a P2P paradigm, research in this regard is still underway. SETI@home [3] comes close to sharing computing power (in the form of idle CPU cycles) of computers connected to the Internet. However, the model employed is still quite centralized. This is because SETI@home allows only a single server to make requests and use idle processing power of other computers in the network. The same capability is not available to all the participants of the SETI@home network. On the other hand, CompuP2P enables all the users to harness almost unlimited processing power of the entire network.

CompuP2P builds and operates dynamic computing resource markets, where sellers and buyers can come together to negotiate transfer/usage of resources from buyer to seller nodes. The lookup of such markets and the availability of resources are robust even in the face of several nodes entering or leaving the network at the same time. CompuP2P uses ideas from game theory ([10]) and microeconomics ([11]) to utilize incentive-based schemes for peers to share their idle computing resources with each other. The pricing strategy is completely distributed without requiring any centralized authority to govern nodes' behavior. For concreteness, in this paper we use compute power as the resource under consideration, however, the mechanism for market creation and resource pricing is equally applicable to any other kind of resource, such as disk space etc.

The rest of the paper is organized as follows. Section II explains our system model. Section III shows how game theory and microeconomics principles can be used to share computing resources in CompuP2P. The potential fault-tolerance issues that may arise in our distributed computation context are discussed in Section IV. Our prototype implementation of CompuP2P, which enables sharing of compute power is described in Section V. Section VI discusses related work on other similar distributed computing projects and we conclude in Section VII with a discussion on future work.

II. Network Model

The network model uses Chord [4] for addressing and nodes' connectivity. We provide a brief description of the Chord protocol in Section II-A. Although CompuP2P uses Chord as the underlying protocol, its architecture is generalized enough such that with little modifications it can also be employed in other structured ([5]) as well as unstructured ([1], [2]) P2P networks.

The network is dynamic as peers join and leave at unpredictable times. Typically in a network there are peers (called *processing nodes*) that may have idle computing resources available to support computing tasks required by other peers (called *clients*). We assume that nodes providing the resources get suitably compensated by the clients. We assume the existence of some electronic payment mechanism as in [6], [7] that is used by clients to compensate the processing nodes.[1]

We assume peers to be selfish, but not malicious. Selfish nodes are rational and strategic in a game theoretic sense, i.e. their intent is to select actions so as to maximize their profits or payoffs and not to cause harm to other nodes or the system in general. The only way for nodes to maximize their payoffs is by selling their idle resources to others that may require them.

The research reported in this paper is funded in part by Jerry R. Junkins Chair position at Iowa State University.

[1]Both the terms ”nodes” and ”peers” are used interchangeably throughout the paper.

A. Chord Overview

Chord [4] supports just one operation, i.e. given a key, it returns the node responsible for that key. Each Chord node has a unique *m*-bit identifier (Chord ID), obtained by say, hashing the node's IP address. Chord views the IDs as occupying a circular identifier space. Keys are also mapped into this ID space, by hashing them to *m*-bit key IDs. Chord defines the node responsible for a key to be the *successor* of that key's ID. The *successor* of an ID *j* is the node with the smallest ID that is greater than or equal to *j* (with wrap-around).

Every Chord node maintains a list of the identities and IP addresses of its *r* immediate successors on the Chord ring. The fact that every node knows its own successor means that a node can always process a lookup correctly: if the desired key is between the node and its successor, the latter node is the key's successor; otherwise the lookup can be forwarded to the successor, which moves the lookup strictly closer to its destination.

In a system with *N* nodes, lookups performed only with successor lists require an average of $N/2$ message exchanges. To reduce the number of messages required to $O(\log N)$, each node maintains a finger table with *m* entries. The i^{th} entry in the table at node *j* contains the identity of the first node that succeeds *j* by at least 2^{i-1} on the ID circle. A new node initializes its finger table by querying an existing node.

III. Markets for Sharing of Computing Resources

In this section we explain how nodes in CompuP2P, possibly across different administrative domains, can share their idle computing resources, specifically compute power. Each node based on its current and past load estimates its average number of CPU cycles that would remain idle in future.[2] 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.[3] These available CPU cycles can be time shared across multiple tasks, as long as the sum of the requirements of all the tasks does not exceed *C*. For example, if *C* is equal to 10^5 cycles/sec, then a node can execute a task that needs at most 10^5 cycles/sec, or if there is no such single task, the processing power may be time-shared between multiple tasks given that the total requirements of the tasks do not exceed 10^5 cycles/sec. 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 computing resources has been estimated, the next step is to determine how to sell them. Moreover, buyers needing extra computing resources should be able to locate the right sellers and purchase the resources from them. The related and equally important issue is how the sellers should price their resources in order to maximize their profits. In the next subsection, we first describe techniques for dynamically creating and locating markets, such that no single node is overburdened with the task of maintaining and running the markets.

[2]For example, by using information from Unix commands, such as "top" and "uptime".

[3]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.

A. Constructing Markets for Buying and Selling Computing Resources

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 successor of Chord ID zero, where all the transactions for all the available compute power in the network take place 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, we propose two schemes that 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 1 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 lies. Thus, nodes with different but close enough processing capacities trade in the same market.

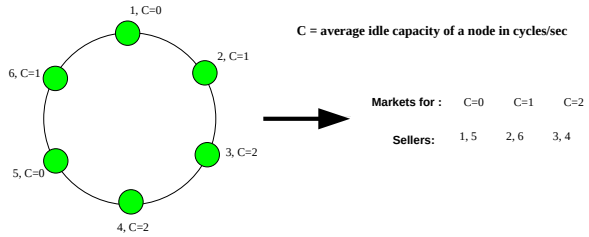

Fig. 1. Creation of markets for CPU cycles in CompuP2P.

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 then become the bottleneck and can degrade the system performance.

2) Processor Overlay Scheme: In order to more uniformly distribute the responsibility for running the compute power markets, 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 value of idle CPU cycles/sec.[4]

The process of selecting a *MO* for a commodity is illustrated in Figure 2. 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 $2^m - 1$).[5] 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 2.

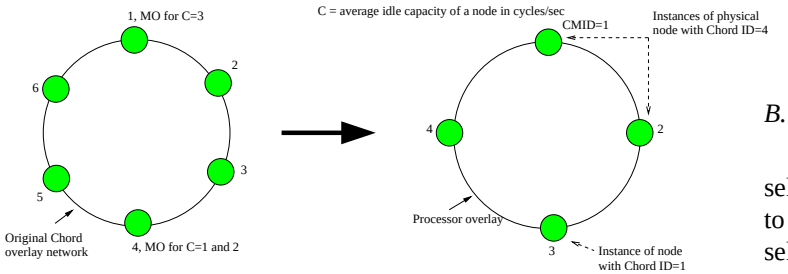

Fig. 2. *Processor overlay* schema using the CPU capacity values given in Fig. 1.

The lookup in processor overlay, requires $\frac{1}{2}(logM)$ steps on average, where *M* is the number of different *CMID* values in the processor overlay network. Moreover, nodes store $O(logM)$ 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 its 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 on the sellers listed in the market.

Nodes have incentive to become *MO*s, since they make profit by charging *listing price* (*LP*) from sellers (and/or buyers) that benefit from the services provided by a market. We describe below two pricing schemes that can be used by a *MO*.

- A *MO* can charge the same fixed price to all the sellers that are listed in the market. This is a simple strategy, however, since there is no central authority to govern the listing price, the *MO* can charge arbitrarily high prices to the sellers and/or may price discriminate among them. Moreover, this scheme also does not take into account the dynamics of a particular market. It seems unfair that sellers should pay the same listing price, when in fact they earn different profits depending on the market they are in and the existing competition. We refer to this scheme as *fixed listing pricing*.
- A *MO* can charge (to the buyers or sellers or both) on the basis of the market characteristics, say some percentage of the selling price. This scheme appears to be fair to both the sellers as well as the *MO*, since a seller is not required to make a payment till it is able to sell its compute power, and the *MO* also potentially gets a higher payoff depending on the dynamics of the market. Although appealing, this scheme is in fact difficult to implement in a distributed setting when the participants (buyers, sellers, and market owners) are all selfish. We refer to this scheme as *variable listing pricing*.

B. Pricing for Computing Resources

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 ([10]) 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

[4]We assume that the value of *c* is large enough to represent the idle processing power of even a very large computer system.

[5]Here we are referring to an existing Chord network comprising of all the nodes, and *m* is the Chord ID size in terms of the number of bits.

than their marginal costs for selling their resources (please see [10] 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.[6] As a consequence, sellers (irrespective of how many there are in a market) in CompuP2P set prices equal to their marginal costs only.

One-shot model of a compute power transaction is reasonable to assume, since once a seller sells its compute power, it delists 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.

The marginal cost of providing computing resources can include among other things - listing price, bandwidth cost for message exchange, etc. and is represented by MC_i for a node, *i*.

1) 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 compute power 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*.

- **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 ([8], [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 ensures 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: Let a seller *A* has the marginal cost (MC_A) of 5 and the lowest marginal cost among all the sellers different from *A* ($= {MC_A}^{-1}$) be 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 than the *MO* receives a low payoff and vice versa. We develop the following simple model for this strategy. Let there be *N* sellers in a market represented by $1, 2, \ldots, N$, such that $MC_i < MC_{i+1}$ for all $1 \leq i \leq N-1$. The sellers are not aware of each other (or 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 proposed 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.

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

  MC'_1 and MC'_N 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.
  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 proposition.
  Proposition 1: Assuming one-shot model of compute power transactions, the payoffs 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.

[6] In CompuP2P all the sellers in a market sell the same amount of a computing resource.

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 $MC_1' + 1 (< MC_1)$. 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 $MC_N' = 2 * MC_1$ (after setting $\frac{\partial \mathrm{Payoff_{MO}}}{\partial \mathrm{MC_N}} = 0$), giving it a payoff of $1/(4 * MC_1)$.[7] Thus, the total cost to the buyer is bounded and is equal to $1/(4 * MC_1) + MC_1 + 1$.
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).

IV. Fault-Tolerant Computing

It is possible that a processing node might not be able to finish the computation assigned to it either because it leaves the network, it 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 processing node's computation, so that if required the failed computations can be migrated to other processing nodes in the network.

Unlike traditional checkpointing, which relies on dedicated checkpoint servers to store the processing state, we propose to use *dynamic checkpointing* in which nodes that store the checkpoint data are determined on-the-fly. Similar to the techniques outlined in Section III 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 may reserve the required memory resources.

Further, in practice errors in computation and/or communication of results can occur. Such errors might be hard to detect and correct. To increase the reliability in the correctness of the end results, one can use redundant computations as also employed in SETI@Home [3]. Basically this scheme involves performing the same computation multiple times at different nodes and then selecting the result produced by the majority of the nodes.

[7]Note that in the given network model, it is difficult for a buyer to verify the marginal cost values it receives from the *MO*.

However, the increased fault-tolerance comes at an increased cost for 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 processing nodes for their compute power.

V. Prototype Implementation for Sharing of Compute Power

We have implemented a Java-based prototype of the proposed CompuP2P architecture for 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. As incentives to the users to sell their idle compute power, we use printing quota as a form of virtual currency, such that users donating more compute power get higher printing quota and vice versa.

A user submits its task to the system in the form of a *task-specification* file. The task-specification file contains a description of a task-tree (describing the inputs and precedence relation among the sub-tasks comprising a task) that needs to be solved, with the following additional information:

- *Code ID*s representing the Java class files and *data keys* for each of the sub-tasks and inputs, respectively. The class files can be downloaded either from a well-defined code server or it can be searched for and downloaded just as other normal data using code ID as the key.
- Estimated amount of compute power required for the sub-tasks.
- User's budget, i.e. the maximum amount of currency that a user can spend in order to get its task successfully completed.

The implementation currently provides for the minimum cost mapping of a task tree to the network nodes. Task tree mappings satisfying other QoS requirements, such as minimum delay or bounded delay with minimum cost are currently not implemented.

We use *SPECjvm98* benchmark [14] to address the problem of nodes' heterogeneity when comparing their compute power. A benchmark program can be selected based on the type of applications typically submitted by the users of the network. Benchmarks help to normalize the compute power values so that a given value is interpreted similarly by all the different nodes. These normalized values help to create homogeneous markets such that different sellers have equivalent compute power to offer, i.e. given a program all the sellers take approximately the same amount of time to execute it. To understand how this normalization is achieved consider the following example. Say, there are two nodes *A* and *B* that takes time T_A and T_B, respectively, to execute certain program *P* of the benchmark. If *A* has C_A and *B* has C_B available compute power, then the normalized idle compute power of *A* and *B* is given by C_A/T_A and C_B/T_B, respectively. These values are then used to determine the market they should join in order to sell their compute power.

Checkpointing as described in the previous section is currently not implemented. We plan to use object persistence

feature provided by PJama ([15]), which would enable a failed computation to be continued at a different node upon failure of an initially allocated processing node.

VI. Related Work

The mechanism for sharing of compute power in CompuP2P is significantly different from other distributed computing projects, such as Condor ([12]), SETI@home, and POPCORN ([16]) that utilize idle processing capacity in the network.

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. Unlike Condor, CompuP2P is completely decentralized, in the sense that there is no centralized entity that create or maintain the markets.

In SETI@home only one central node can allocate tasks to others, whereas in CompuP2P all the peers can purchase computing power and distribute their workload onto other peers in the system.

Both Condor and SETI@home do not take into account nodes' selfishness and assume that nodes provide their compute power without requiring any form of compensation in return. For example, SETI@home appeals on the participants' altruism to contribute their resources in the quest to search for life in outer space.

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 [13] that relies on centralized auctioning.

VII. Discussion and Future Work

Our current implementation of CompuP2P that permits sharing of compute power can be used to build distributed processing systems that can potentially reduce (or eliminate) the need for large and expensive processing servers in an enterprise. Users of CompuP2P can harness almost unlimited processing power of the entire network.

In this paper we have described mechanisms for creation of markets and pricing of computing resources in a completely decentralized and robust manner. These mechanisms take into account nodes' selfishness without relying on any trusted centralized authority. However, in order to fully realize the potential of CompuP2P and implement it for real-world applications, several issues still need to be resolved.

First, CompuP2P relies on a monetary payment scheme to compensate processing nodes for their 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, we are developing mechanisms for using reputation as a form of virtual currency instead.

Second, in any large-scale decentralized network there is a possibility of malicious nodes, which we have ignored till now in our discussion. Malicious nodes can complicate the pricing of computing resources. For example, ideally, a client wants to pay only for the completed and correct computation results. However, a malicious node may deliberately generate wrong results and/or to save its compute power may give out only partial results to the client. We refer to such incorrect or incomplete computations simply as *faults*. The types of faults that can be generated would depend on the nature of tasks being processed. For most applications such faults can be hard to detect and it is even more difficult to prove that they were deliberately introduced by a processing node. In such scenarios how much (if at all) the processing nodes should be paid is a tricky question.

We feel that the reliability of the received computation results can be improved by distributing a computation to multiple processing nodes and select the output generated by the majority of the nodes. Then the nodes whose results do not conform with the majority results are not compensated by the client. These and other possible solutions are part of our on-going investigation. We feel that the precise solutions employed would be application-domain dependent and hope that our efforts in using CompuP2P for implementing large-scale simulations in our lab would provide us with valuable insights into studying fault-tolerance for distributed computation in CompuP2P.

References

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

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

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

[4] 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.

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

[6] G. Medvinsky. A Framework for Electronic Currency. *PhD thesis, USC*, 1997.

[7] M. Bellare, J. Garay, C. Jutla, and M. Yung. VarietyCash: a multi-purpose electronic payment system. *In Proc. Of 3rd Usenix Workshop on Electronic Commerce, pages 9-24*, August 1998.

[8] N. Nisan. Algorithms for Selfish Agents: Mechanism Design for Distributed Computation. *In Proceedings of the 16th Symposium on Theoretical Aspects of Computer Science, Lecture Notes in Computer Science, volume 1563, Springer, Berlin, pages 1-17*, 1999.

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

[10] M. J. Osborne. An Introduction To Game Theory. New York, Oxford : Oxford University Press, 2004.

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

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

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

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

[15] The PJama Project. http://www.dcs.gla.ac.uk/pjava/.

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