

# CompuP2P: An Architecture for Large-Scale Ad-Hoc Grids Based on Peer-to-Peer Paradigm

## Abstract

CompuP2P is an architecture for large-scale dynamic ad-hoc grids based on peer-to-peer paradigm that provide computation capabilities to processing-intensive user applications. CompuP2P uses ideas from game theory to govern sharing of idle compute power among the grid nodes. Dynamic markets for compute power are created that facilitate migration of tasks from clients requiring processing to nodes that are currently lightly loaded. A Java based prototype of the proposed architecture has been developed that enables user to execute large and complicated tasks by specifying their task requirements in the form of a task tree.

## Index Terms

Peer-to-peer, ad-hoc grids, distributed computing, task tree, game theory.

## I. INTRODUCTION

**C**OMPUP2P is an architecture for large-scale dynamic ad-hoc grids based on peer-to-peer (P2P) paradigm that provide computation capabilities to processing-intensive user applications. Much of the focus of grid computing to date has assumed the existence of static grid sites, which have out-of-band trust relationship among themselves. However, there is a need among small workgroups and laboratories for a less formal and organized grid environment. Such an ad-hoc computing environment would eliminate the need for grid infrastructure administration and instead offer a more peer-to-peer type grid environment to users. In the paper, we propose a self-organizing and self-governing grid architecture that allow users to execute large tasks by utilizing the aggregate compute capacity of the grid nodes.

CompuP2P enables efficient utilization of network resources, such as bandwidth by merging and pruning data before it is sent to the receiver. CompuP2P allow computationally (and memory) limited devices, such as wireless hosts (e.g. PDAs) to access network services by using other nodes as their agents.<sup>1</sup> Multiple grid nodes form a virtual private processing system by contributing their individual resources, such as idle CPU cycles. Such a system can then perform compute intensive tasks, such as scientific simulations or data mining on behalf of the requesting clients. Applications, like scientific simulations and data mining, requiring huge processing can tremendously benefit from this potentially unlimited availability of idle compute power.

The goal of CompuP2P is to build completely decentralized Internet-scale ad-hoc grid networks utilizing the processing power and other resources, such as memory of millions of PCs and workstations that have

Technical Report

<sup>1</sup>Generally speaking services as used here refer to tasks that depend on multiple data objects that need to be processed in certain specified order in order to generate meaningful results.

Internet connectivity and are under-utilized most of the time. Users of CompuP2P can harness almost unlimited processing power of the entire network. The architecture for distributed processing uses the Chord [4] protocol for locating data that constitute input to processing. CompuP2P construct globally efficient mapping of users' task trees to grid nodes, such that the QoS constraints specified by the users are satisfied. Some of the QoS constraints associate with a distributed computation can be minimum cost, minimum delay, or minimum cost with bounded delay.

CompuP2P builds and operates dynamic compute power markets, where sellers and buyers can come together to negotiate transfer of processing tasks from buyer to seller nodes. The lookup of such markets and the availability of processing power are robust even in the face of several nodes entering or leaving the network at the same time. CompuP2P uses ideas from game theory ([12]) to develop incentive-based schemes for nodes to share their idle compute power with each other. The pricing strategy for compute power is completely distributed without requiring any centralized authority to govern nodes' behavior.

The rest of the paper is organized as follows. Section II explains our network model. Section III shows how game theory principles can be used to share compute power in CompuP2P. Section IV explains how task trees can efficiently be mapping to grid nodes, with a worked out example in Section V. In Section VI we propose some schemes for building fault-tolerance into large scale distributed computations in the context of CompuP2P. Section VIII talks about related work. Section VII describes our Java implementation of CompuP2P and we conclude the paper in Section IX.

## II. NETWORK MODEL

A large grid consist of nodes (called *processing nodes*) may have idle compute power that can be used by some other nodes (called *clients*). The network model assumes a P2P configuration that uses Chord [4] for data lookups and nodes' connectivity. The system is dynamic as nodes join and leave at unpredictable times.

Peer-to-peer networks ([1], [2], [5], [4]) are flexible distributed systems that allow nodes (also called peers) to act as both clients and servers to access 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.

A client node requiring some computation service ( $S$ ) can request others to carry out the desired tasks

for the service completion.<sup>2</sup> We assume that nodes providing the compute power get suitably compensated by the clients for their CPU cycles. To simplify our discussion, we do not explicitly consider compensating nodes for their data, however, in [7] we have proposed a pricing strategy in order to motivate nodes to share their data with each other. We assume the existence of some electronic payment mechanism as in [8], [9] that is used by the clients to compensate the processing nodes.

We assume all the grid nodes to behave selfishly. This assumption is important as nodes belong to independent individual users. Selfish nodes are strategic and rational in a game theoretic sense, i.e. their intent is to maximize their own profits or payoffs. The only way for nodes to maximize their payoffs is by selling their idle compute power to others that might require them.

We use the notation  $D_i$  to refer to both a data type and its specific instance called a data object. A nodes service requirement can be represented by a vector:  $S = \cup D_i$ , implying that it relies on several data objects (of data type  $D_i$ , respectively) that need to be processed in a certain specified order. For example, a client needs to perform service  $S$  described by,

$$S = (D_1 \oplus_1 D_2) \oplus_3 (D_3 \oplus_2 D_4). \quad (1)$$

We refer to  $D_i$  and  $\oplus_i$  as tokens that indicate data and operation types, respectively. Thus service  $S$  requires performing operation  $\oplus_1$  on objects  $D_1$  and  $D_2$ ,  $\oplus_2$  on objects  $D_3$  and  $D_4$  and then computing  $\oplus_3$  on the results of the two sub-operations or subtasks. Each of the different operations may require different program code, inputs etc., and hence require different amount of compute power for carrying them out. A client can determine the compute power that would be required for each of the operations. The order of processing of various input data and intermediate results is important in a complex task where the subtasks may be arbitrarily nested. For simplicity we assume the use of binary operations only. Other complex operations can be formed using the binary operators. Unary operators are treated as special forms of binary operators in which one of the operand has a "zero" value.

A node that stores an index for some data, as defined by Chord, is also the *data pool manager* ( $DP_i$ ) for that data type. We provide a brief introduction to Chord in Section II-A. Several nodes may have stored a copy of some data, each instance of which is referred to as a data object. For example, copies of a file may be stored at multiple places through replication and/or caching.  $DP_i$  stores the Chord ID and IP address of all the nodes which have a copy of the data corresponding to  $D_i$ . A distinguishing feature of caching

<sup>2</sup>A client can include a wireless host, such as a PDA or a laptop with a low battery power and intermittent network connectivity.

in CompuP2P is that a node on receiving a lookup request, even if it has a copy of the data object, still forwards it to the  $DP_i$  node (in one logical hop). This simple extension allows the request to be fulfilled by the data pool manager in an optimal manner given the current distribution of data around the network.

The model of distributed computing on which CompuP2P is based is frequently encountered in real-world applications as suggested by the following examples. These and several other similar examples of distributed computing for commercial and scientific applications are described in detail in [6].

- *Task-parallel applications*: They are so called because parallelism is achieved by partitioning the work to be performed rather than the data to be operated on. A full-fledged task-parallel programming model must also include functionality for collecting and combining results from application tasks and for automatically balancing the load for improved performance. The overwhelmingly popular paradigm for task parallelism is the master-worker model in which a master process divides portions of the entire workload among populations of worker processes. The master-worker paradigm is well suited to CompuP2P, as application tasks need not be tightly coupled. Master-worker has been used for challenging computations, including the solution of open problems in numerical optimization.
- *Parameter Sweep Applications (PSAs)*: PSAs comprise sets of computational tasks that are mostly independent, although there are few task synchronization requirements or data dependencies among tasks. This simple application model arises frequently in many fields of science and engineering, including computational fluid dynamics, bioinformatics, particle physics, discrete-event simulation, computer graphics, astronomy, and computational biology. PSAs are attractive as CompuP2P applications because they are not tightly coupled: tasks do not have stringent synchronization requirements and can thus tolerate high network latencies. In addition, they are amenable to straightforward fault-tolerance mechanisms such as restarting tasks from scratch after a failure.
- *Workflow Applications*: Workflows consist of modules and data filters with well-specified input and output. The modules are connected in order to achieve some desired goal. Workflows arise in many contexts. Several problem solving environments (PSEs) that enable users to build and execute large workflows have been developed. It is typical for workflows to be so large, and/or application data and participants to be so distributed, that it is not feasible to run the entire application within a single system or institution. Thus, there is a natural affinity between workflow applications and CompuP2P.

### 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. MARKET MODEL FOR SHARING OF COMPUTE POWER

In this section we explain how nodes in CompuP2P, possibly across different administrative domains, can share their idle compute power. Each node based on its current and past load estimates its average number of CPU cycles that would remain idle in future.<sup>3</sup> 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. These available CPU cycles can be time shared across multiple tasks, as long as the sum of the requirements of all the tasks do 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.

<sup>3</sup>For example, by using information from Unix commands, such as "top" and "uptime".

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 processing power from them. The related and equally important issue is how the sellers should price compute power 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.

#### A. Constructing Markets for Buying and Selling Compute Power

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.

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,

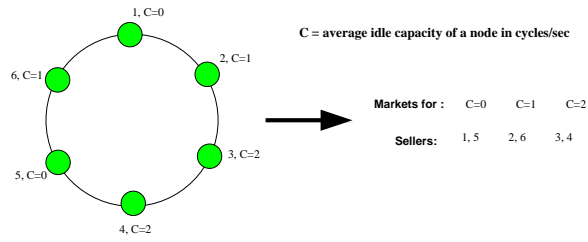


Fig. 1. Creation of markets for CPU cycles in CompuP2P.

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 *MOs*, which are responsible for various commodities, constitute this Chord-based overlay network. The total ID space of this new overlay is equal to the maximum amount of compute power that may possibly be available on any single node and is upper-bounded by  $2^c - 1$ , where  $c$  is a constant and represents the number of bits used to represent the value of idle CPU cycles/sec.<sup>4</sup>

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  $2^m - 1$ .<sup>5</sup> The successor node of  $hash(C)$  is then the *MO* for the market trading in commodity  $C$ . The various *MOs* defined in this manner then together form another overlay network, called the *processor overlay*, which has ID space from 0 to  $2^c - 1$ . The ID of a *MO* in this new overlay network is simply the value  $C$  whose hash value was mapped to it in the initial Chord network. Stated otherwise, the ID of a *MO* in the processor overlay network, called CPU Market ID (*CMID*), is the number of CPU cycles/sec that are being sold in its market.

It must be noted that in the above description, it is possible that a single node in the initial overlay network

<sup>4</sup>We assume that the value of  $c$  is large enough to represent the idle processing power of even a very large computer system.

<sup>5</sup>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.

is the *MO* for several different markets, causing it to have multiple *CMIDs* assigned to it in the processor overlay network. Each *CMID* value is represented by a different node in the processor overlay as shown in Figure 2.

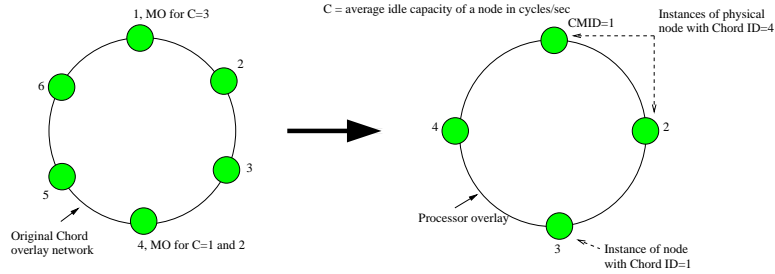


Fig. 2. *Processor overlay* schema using the CPU capacity values given in Fig. 1.

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

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 *MOs* and forwards the lookup request to it. The selected *MO* searches for an appropriate market for the desired compute power in the processor overlay network. The lookup process finally returns the IP address of the *MO* that runs the market for that compute power or the nearest higher compute power value available in the network. For example, if only two compute power markets (with commodity values  $b$  and  $c$ ) exist in the network, and a client desires  $a$  (where  $a < b < c$ ), then the above mechanism returns market for  $b$  instead of  $c$ . The *MO* is then contacted to obtain information on the sellers listed in the market.

Nodes have incentive to become *MOs*, 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, it is difficult for different *MOs* to coordinate and come up with a single price that is applicable to all the different markets. 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 (possibly zero) 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 Compute Power

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 (the classical Prisoner's dilemma problem [12] and Bertrand oligopoly [13]), 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 compute power. In Bertrand oligopoly sellers strategy is to set "prices" (as opposed to "outputs" in Cournot oligopoly) and is thus more reasonable to assume in the context of CompuP2P.

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.

The marginal cost of providing compute power 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 ([10], [11]). 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 [11] 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  $1, 2, \dots, N$ , such that  $MC_i < MC_{i+1}$  for all  $1 \leq i \leq N - 1$ . We assume all the marginal costs to be integer values. 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 payoff functions used by a buyer to compensate the *MO* and the selected seller (represented by  $Payoff_{MO}$  and  $Payoff_{seller}$ , respectively) are well-known and are given as follows.

$$\begin{aligned} Payoff_{MO} &= (MC'_N - MC'_1)/(MC'_N)^2 \\ Payoff_{seller} &= MC'_1 + 1 \end{aligned} \tag{2}$$

$MC'_1$  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 2 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  $I$ ) 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 2 where we see that the payoff received by the seller is one more than its marginal cost.

c) From Equation 2, the payoff to the  $MO$  is maximized for  $MC'_N = 2 * MC_1$  (after setting  $\frac{\partial \text{Payoff}_{MO}}{\partial MC_N} = 0$ ), giving it a payoff of  $1/(4 * MC_1)$ .<sup>6</sup> 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 2.

■

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. MAPPING TASK TREE TO GRID NODES

After having looked at how idle compute power can be efficiently searched and traded in a large grid network, we next explain how a task is executed in CompuP2P in a distributed manner. Assume that a client node needs to perform some computation or service  $S$ , which is specified by the task Equation 1. Such a computation can also be represented in the form of a task tree as shown in Figure 3. The tree representation captures the precedence relationship among the various tokens in a task equation.

A client gathers the list of nodes that are capable of processing the tokens in its task equation. This is done by contacting the data pool managers and  $MO$ s for the data and operation tokens, respectively. When contacting a  $MO$ , a client can specify whether it wants the lowest cost seller or a list of all the available sellers in the market. A complete list of sellers is requested in order to select the node that best meets the QoS requirements of the computation. If sufficient compute power is not available for any operation token, a client aborts the service computation and retries at a later time. The lists of nodes obtained are placed at the corresponding nodes of the task tree and the resulting tree is called the expanded task tree ( $E-TT$ ) as

<sup>6</sup>Note that in the given network model, it is difficult for a buyer to verify the marginal cost values it receives from the  $MO$ .

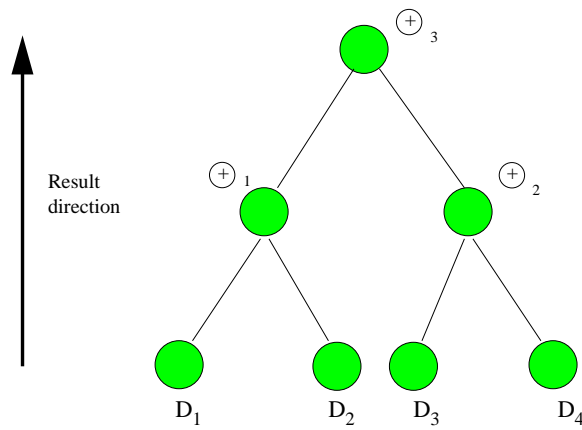


Fig. 3. Task Tree corresponding to service  $S$ .

shown in Figure 4. A node in  $E-TT$  is referred to as the *token set (T-Set)*, since it is a set of addresses of physical nodes that are capable of processing the corresponding token.

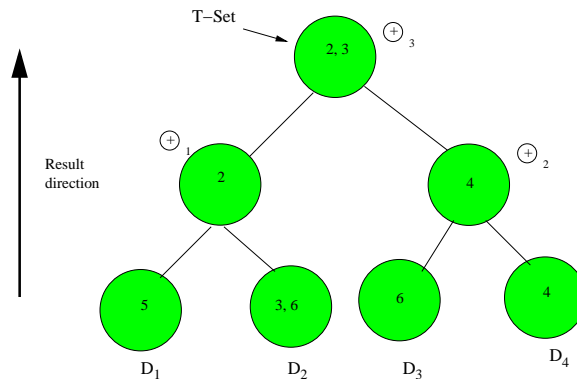


Fig. 4. Expanded Task Tree (E-TT) containing the list of nodes capable of processing the various tokens represented by the Task Tree of Figure 3.

### A. Centralized Token Allocation

It is possible that there are multiple nodes that can process each token in a task equation, i.e. there may be multiple nodes that have a copy of the data and also multiple nodes that are willing to provide the desired compute power. In centralized task allocation, the goal is to allocate tokens, such that the overall task (or service) can be performed efficiently, i.e. satisfy the client's QoS requirements.<sup>7</sup> This process can be understood as the mapping of a client's task tree onto a subset of grid nodes. We assume that clients use one or more of the following service QoS requirements - minimum cost, minimum delay, or minimum cost with bounded delay.

<sup>7</sup>Allocation of data token means selecting nodes that supply data and allocation of operation token refer to selecting nodes where processing of data occur.

If a client is looking to minimize the total cost of its computation, it can select the minimum cost supplier from the markets for each of the different operation tokens using the payoff strategy described in Section III-B. For satisfying other QoS requirements, a client obtains information, including the  $MC$  values, on all the sellers in a market corresponding to every operation token in its task equation. The pricing strategy, which determine the payoffs to the  $MOs$  and sellers, in such cases is not obvious and is part of our ongoing investigation.

Below we first describe how the mapping of a task tree onto grid nodes can be carried out such that the overall time for computation is minimized.<sup>8</sup> The proposed algorithm assumes the existence of a mechanism (see [14], [15]) that enables finding relative physical distances between any pair of nodes in a network.

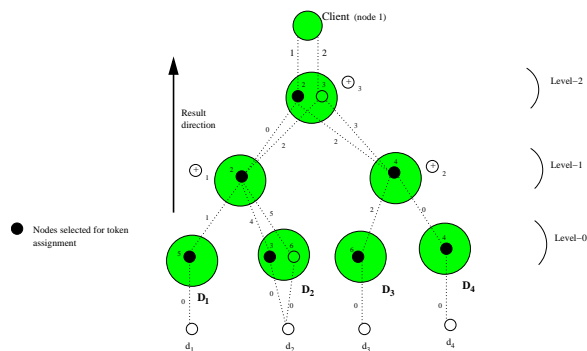


Fig. 5. Topology based expanded task tree ( $TBE-TT$ ).

We augment the  $E-TT$  by transforming every pair of parent-child  $T-Sets$  into complete bi-partitions, such that link weights correspond to the shortest path distances between elements of the two  $T-Sets$ . Moreover, for each  $T-Set$  at Level-0 (i.e. height 0 of the binary tree) a dummy node  $d_i$  is added that is connected by zero cost links to the elements of the corresponding  $T-Set$ . Furthermore, a new root node representing the client node is added. The link weights again correspond to the physical distance between the client node and the nodes representing the elements of the original root  $T-Set$ . The augmented  $E-TT$  is referred to as the *topology based expanded task tree* ( $TBE-TT$ ) and is shown in Figure 5.

In centralized token allocation the goal is to select a physical node, i.e. an entry from each of the  $T-Set$ , such that the overall computation can be performed in minimum time without exceeding the capacity of any single node. Equivalently, given a  $TBE-TT$ , we need to determine the shortest path from the root (i.e. the client) to all the leaf nodes (i.e.  $d_i$ s). The resulting shortest path should minimize the total time to reach

<sup>8</sup>The minimum time criteria not only satisfies the service delay requirements, but also minimizes network resource usage since the overall path of computation (i.e., the total distance travelled by the intermediate results) is minimized.

from the bottom to the top of the tree.<sup>9</sup> In other words, the sum of the maximum of the link weights between adjacent levels must be minimized. In addition, the total processing capacity of any physical node included in the resulting solution should not be exceeded. This can happen because the same node(s) that is part of multiple *T-Sets* may get selected for processing operation tokens whose total processing requirements may exceed the node's capacity. The shortest path that satisfy the above constraints is referred to as the *constrained shortest path*.

In Appendix A, we prove that the problem of finding the constrained shortest path in *TBE-TT*, henceforth simply referred to as the *TBE-TT* problem, is in fact NP-complete. The proof relies on showing the *TBE-TT* problem to be an instance of the set-cover problem [22]. Therefore, we exhaustively search all the available paths in *TBE-TT* from the client to the  $d_i$ s and select the one that minimizes the computation time and also meets the processing capacities of the selected nodes.

For example, the shortest path computed in Figure 5 assigns token  $D_1$  to node 5,  $D_2$  to node 3,  $\oplus_3$  to node 2, and so on. Token assignment for the case when a buyer desires minimum cost with bounded delay can similarly be solved by finding a least cost path among all the feasible paths (i.e., which satisfy the delay bound).

## V. DISTRIBUTED COMPUTATION STEPS IN COMPUP2P

In this section we consider a simple example to summarize as to how a distributed task is executed in CompuP2P. The table in Figure 6 is an example of the system state at some time instant. The values of the various entries in the table are selected so as to simplify our discussion here. For illustration, node 3 has available compute power of 1150 that it can provide (for  $T$  time units) if it receives a payoff of at least 15. In addition, it is a *MO* for compute power in the range [1, 100] and has a copy of data represented by  $D_2$ . It is assumed that the compute power values that are within an interval of 100 trade in the same market, i.e. [1, 100], [101, 200], and so on define the bounds such that all compute power values in these intervals trade in the same market.

Suppose node 1 has a service requirement represented as,  $S = (D_1 \oplus_1 D_2) \oplus_3 (D_3 \oplus_2 D_4)$ . We give the steps carried out by node 1 to accomplish  $S$  in minimum possible time.

- Node 1 contacts the data pool manager  $DP_i$  ( $\forall i \in \{1, 2, 3, 4\}$ ) to get information on nodes that can provide data,  $D_i$ .

<sup>9</sup>Although processing delays at nodes are not discussed, such delays can be taken into account by simply adding them to the link delays.

Nodes (Chord ID)	Available compute power	Marginal cost (MC)	Market owner (MO) for capacities in the range	Data stored (cached or replicated)
1	10	24	–	–
2	1110	15	–	–
3	1150	15	1–100	$D_2$
4	250	24	1101–1200	$D_1$
5	14	25	–	$D_1$
6	20	30	201–300	$D_2, D_3$

Fig. 6. Network snapshot with six nodes.

- Using the meta information received about the data objects (such as their size etc.) and the nature of processing to be done on them, node  $I$  estimates the amount of compute power that would be required for each of the operation tokens, as calculated in Figure 7.

Operation token	CPU cycles required (per sec)
$\oplus_1$	500
$\oplus_2$	200
$\oplus_3$	600

Fig. 7. Processing requirements of node  $I$  for service  $S$ .

- After determining the processing requirements, node  $I$  performs a lookup for the desired amount of compute power in the *processor overlay* for each of the operation token. In Figure 8 it can be seen that

Operation token	Potential suppliers of compute power
$\oplus_1$	2, 3
$\oplus_2$	4
$\oplus_3$	2, 3

Fig. 8. Nodes with sufficient idle compute power to process the operation tokens.

the lookup for both  $\oplus_1$  and  $\oplus_3$  lead to the same *MO* (node 4) and return the same sellers (nodes 2 and 3).

- After performing the necessary information collection steps, node  $I$  creates a *topology based expanded task tree (TBE-TT)* as shown in Figure 5. The numbers next to the links are the distances between the

nodes. In order to minimize the total execution time for  $S$ , node  $I$  does an exhaustive search to find the minimum weight path from node 1 to nodes  $d_1, d_2, d_3$ , and  $d_4$ . The result is the set of nodes that are selected for executing the various data and operation tokens comprising service  $S$ . The mapping of task tree onto the grid nodes is shown in Figure 9. The next step is resource reservation and connection setup such that the selected processing nodes cooperate to execute various subtasks in order to generate the final result and send it to the client node.

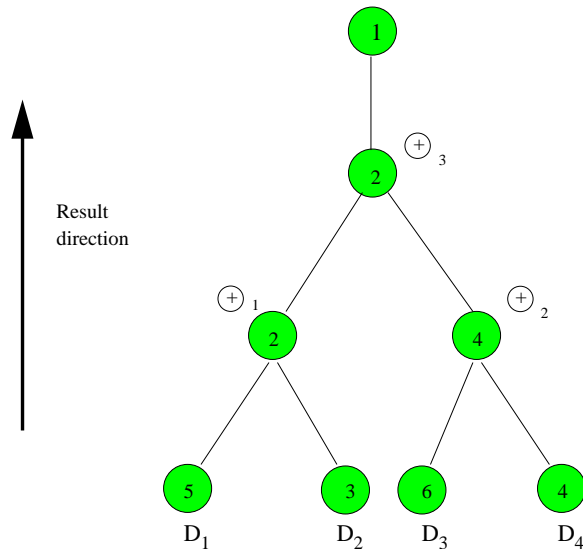


Fig. 9. Mapping of task tree for  $S$  onto grid nodes.

## VI. 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 generated by the operation tokens may reserve the required memory resources. The amount of memory to be reserved is the maximum of the amount of checkpoint data that is generated by computations at any level of the task

tree. For example, in Figure 9 memory is required only to store checkpoint data generated by computations at Level-1 and not for both Level-1 and Level-2. This is because computation at Level-2 cannot begin unless they are finished at Level-1, and therefore checkpoint data generated by processing node 2 at Level-2 can re-use the space allocated for Level-1 computations.

Further, in practice errors in computation and/or communication of intermediate results can occur. Moreover, nodes may act maliciously and deliberately produce incorrect results for the computation assigned to them. All 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. In CompuP2P, such redundant computation is achieved by constructing several mappings of a task tree and then selecting the result generated by the majority of the mapped computations.

However, the increased fault-tolerance comes at increased cost for the user. The user 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.

## VII. PROTOTYPE IMPLEMENTATION

We have implemented a Java-based prototype of the proposed CompuP2P architecture 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 the description of a task tree that needs to be solved and includes the following information:

- *Code IDs* representing the Java class files and *data keys* for each of the operation and data tokens, 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 operation tokens.
- User's budget, i.e. the maximum amount of currency that the 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 grid 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 [16] 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 grid. 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 take 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 and we plan to use object persistence feature provided by PJama [17], which would enable a failed computation to be continued at a different node upon failure of an initially allocated processing node. While selecting a new processing node, care must be taken to ensure that the initial cost and/or QoS constraints associated with a computation are not violated.

## VIII. RELATED WORK

CompuP2P is significantly different from other previous distributed computing projects, such as Condor [19], SETI@home [3], and POPCORN [18], which were also developed with the objective of harnessing idle CPU cycles 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 monitors system state and assign (sub)tasks accordingly.

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 [20] that uses a centralized auction.

## IX. CONCLUSION AND FUTURE WORK

CompuP2P can be used to build large ad-hoc grids for distributed processing that can potentially reduce the need for expensive processing servers in an enterprise, for example. 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 compute power in a completely decentralized and robust manner. These take into account nodes' selfishness without relying on any trusted centralized authority. CompuP2P relies on a monetary payment scheme to compensate processing nodes for their compute power. 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 [21] we have proposed a framework for using reputation as a form of virtual currency instead.

The proposed pricing strategy works fine when computations proceed correctly, however, compensating processing nodes in case failures or incorrect results becomes tricky and we would explore such issues as part of our future work. Moreover, we would develop appropriate heuristics for the *TBE-TT* problem without relying on exhaustive search, such that efficient mapping of task trees onto grid nodes can be carried out satisfying different QoS requirements.

## 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] I. Foster, and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*, 2nd Edition, Morgan Kaufmann, 2004.
- [7] R. Gupta, and A. K. Somani. An Incentive Driven Lookup Protocol For Uncooperative Peer-to-Peer (P2P) Networks. *Submitted to ICON 2004*. (A copy of the submitted draft can be obtained from the following URL: [http://ecpe.ee.iastate.edu/dcnl/DCNLWEB/Publications/pub-research\\_tech-rep.htm](http://ecpe.ee.iastate.edu/dcnl/DCNLWEB/Publications/pub-research_tech-rep.htm))

- [8] G. Medvinsky. A Framework for Electronic Currency. *PhD thesis, USC*, 1997.
- [9] 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.
- [10] 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.
- [11] W. Vickrey. Counterspeculation, auctions and competitive sealed tenders. *Journal of Finance, pages 8-37*, 1961.
- [12] M. J. Osborne. A course in game theory. Cambridge, Mass. : MIT Press, c1994.
- [13] M. R. Baye. Managerial Economics and Business Strategy. *Third edition, McGraw Hill*, 2000.
- [14] T. S. Eugene, and H. Zhang. Predicting Internet Network Distances with Coordinates-Based Approaches. *INFOCOM 2002, New York, NY, June 2002*.
- [15] V. N. Padmanabhan, and L. Subramanian. An Investigation of Geographic Mapping Techniques for Internet Hosts. *In Proc. of ACM SIGCOMM '01, San Diego, USA, Aug. 2001*.
- [16] Standard Performance Evaluation Corporation. *SPECjvm98 Documentation*, Release 1.0. August 1998. Online version at <http://www.spec.org/osg/jvm98/jvm98/doc/index.html>
- [17] The PJama Project. <http://www.dcs.gla.ac.uk/pjava/>.
- [18] 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.
- [19] P. Wagstrom. An Overview of Condor. February 19, 2002.
- [20] M. Senior, and R. Deters. Market Structures in Peer Computation Sharing. *Second International Conference on Peer-to-Peer Computing (P2P'02)*, 2002.
- [21] R. Gupta, and A. K. Somani. A Framework for Reputation Management and Using Reputation as Currency in Large-Scale Peer-to-Peer Networks. *To appear in P2P 2004*.
- [22] P. Slavik. A Tight Analysis of the Greedy Algorithm for Set Cover. *STOC '96, Philadelphia, PA, USA, 1996*.

## APPENDIX A

There are  $N$  physical nodes,  $n_1, n_2, \dots, n_N$ , with capacities (in cycles/sec)  $C_1, C_2, \dots, C_N$ , respectively. Some or all of these nodes constitute the various  $T$ -Sets of a  $TBE$ - $TT$ . It is possible to have instances of a node in multiple  $T$ -Sets. Each  $T$ -Set represents a token in the task equation and one node out of a  $T$ -Set is selected for the corresponding token processing.

We define an instance ( $\zeta$ ) of the  $TBE$ - $TT$  problem as follows. Let  $G = (V, E)$  represent a  $TBE$ - $TT$ , where  $V$  is the set of vertices (representing  $T$ -Setss) and  $E$  is the set of edges.  $G$  is a complete binary tree (except at the top most level) with unit edge weights (denoted by  $w$ ), i.e.  $w(e) = 1, \forall e \in E(G)$ .<sup>10</sup> The  $TBE$ - $TT$  problem is to find a constrained shortest path (CSP) from the root (client) to all the leaf (dummy) nodes. Without loss of generality, the  $T$ -Sets in  $G$  can be represented by  $T = \{1, 2, \dots, |V|\}$ . Then the CSP is a vector  $\vec{K}$  ( $|\vec{K}| = |V|$ ), called the  $CSP$ -vector, whose  $i^{th}$  element denotes a physical node to which the token corresponding to  $T$ -Set  $i$  is allocated. Below we give the steps for obtaining a  $CSP$ -vector.

- Derive the different *feasibility sets* for all the physical nodes. The feasibility set for a physical node is a set of all the combinations of tokens that it can process without exceeding its capacity. For example, if there are three tokens corresponding to  $T$ -Sets 1, 2, and 3, respectively, and node  $n_1$  cannot process 1 and 2 simultaneously, then the resulting feasibility set for  $n_1$  would be,  $T_{n_1} = \{\{1\}, \{2\}, \{3\}, \{1, 3\}, \{2, 3\}\}$ .

<sup>10</sup>The weights here refer to the physical distances between the nodes. For simplicity we assume that in the underlying physical network, the distance between any pair of distinct nodes is unity.

The  $j^{\text{th}}$  element of the feasibility set is represented as  $T_{n_l}^j$  for  $j = 1, 2, \dots, |T_{n_l}|$ . In general,  $T_{n_l}^j \subseteq T, \forall l \in \{1, 2, \dots, N\}$ .

- Find the minimum number of these  $T_{n_l}^j$ , together represented by  $\tau$ , such that  $\cup T_{n_l}^j = T$ .
- The *CSP-vector* can be  $\{n_l | \exists j T_{n_l}^j \in \tau\}$ .

*Theorem 1:* The *TBE-TT* problem is NP-complete.

*Proof:* We first show that  $\zeta \in \text{NP}$ . Let there be a certificate  $[k_1(c_1), k_2(c_2), \dots, k_{|V|}(c_{|V|})]$  and a weight  $W$ , where  $k_i$  ( $k_i \in \{n_1, n_2, \dots, n_N\}$ ) represents the  $i^{\text{th}}$  element of a *CSP-vector* to which the token corresponding to *T-Set*  $i$  is allocated and  $c_i$  is the compute power required to process that token. Same physical node might appear more than once in the certificate. We can verify that indeed the token allocation indicated by the certificate is correct by ensuring that it does not cause the capacity of any single node to be exceeded and that the total time for completing the computation is less than or equal to  $W$ . This verification can be performed in polynomial time. Thus,  $\zeta \in \text{NP}$ .

Next we show that the *TBE-TT* problem is NP-hard by showing that minimum set-cover  $\leq_p \zeta$ . Minimum set-cover ([22]) is a NP-complete problem.

Let us define an instance of the minimum set-cover problem, referred to as  $SC$ , which is constructed as follows. Let  $S = s_1, s_2, \dots, s_{|V|}$  be the ground set. Also, let set  $S_l$  be equal to  $S^* - R_l$ . Here,  $S^*$  is a set of all possible subsets of  $S$  (total  $\sum_{i=0}^N \binom{N}{i}$  in number) and  $R_l$  ( $R_l \subseteq S^*$ ) is the *restraint set* for node  $n_i$ . A restraint set include combinations of elements of  $S$  that cannot belong to  $S_l$ . Elements of  $S_l$  can be represented as  $S_l^j$ , where  $j = 1, 2, \dots, |S_l|$ .  $SC$  can now be defined as finding the minimum number of these  $S_l^j$ , such that  $\cup S_l^j = S$ .

We map  $SC$  to  $\zeta$  in polynomial time as follows. We set  $T = S$ , and  $T_{n_l} = S_l, \forall l \in \{1, 2, \dots, N\}$ . Thus,  $SC$  maps to  $\zeta$  by a direct one-step transformation. Likewise,  $\zeta$  can be transformed to  $SC$  in polynomial time. We can see that a solution for  $\zeta$  will also be a solution for  $SC$  (and vice versa). Since, no polynomial time solution is currently known for  $SC$ , there is likely none for  $\zeta$ . Hence the *TBE-TT* problem is NP-complete. ■