An Incentive-Based Distributed Mechanism for Scheduling Divisible Loads in Tree Networks

T.E. Carroll\textsuperscript{a}, D. Grosu\textsuperscript{b,*}

\textsuperscript{a}Pacific Northwest National Laboratory, P.O Box 999 MSIN J4-45, 902 Battelle Boulevard, Richland, Washington 99352 USA
\textsuperscript{b}Department of Computer Science, Wayne State University, 5057 Woodward Avenue, Detroit, Michigan 48202 USA

Abstract

The underlying assumption of Divisible Load Scheduling (DLS) theory is that the processors composing the network are obedient, i.e., they do not “cheat” the scheduling algorithm. This assumption is unrealistic if the processors are owned by autonomous, self-interested organizations that have no \textit{a priori} motivation for cooperation and they will manipulate the algorithm if it is beneficial to do so. In this paper, we address this issue by designing a distributed mechanism for scheduling divisible loads in tree networks, called DLS-T, which provides incentives to processors for reporting their true processing capacity and executing their assigned load at full processing capacity. We prove that the DLS-T mechanism computes the optimal allocation in an ex post Nash equilibrium. Finally, we simulate and study the mechanism under various network structures and processor parameters.

1. Introduction

One of the most studied topics in distributed systems is scheduling. Poor scheduling decisions lead to inefficiencies, underutilized resources, and suboptimal performance. In this paper, we focus on the problem of scheduling divisible loads. Divisible load problems are characterized by data parallelism. These problems have large data sets where every element within the set requires an identical type of processing. The sets

\textsuperscript{*}Corresponding author

Email addresses: Thomas.Carroll@pnl.gov (T.E. Carroll), dgrosu@wayne.edu (D. Grosu)
can be partitioned into any number of fractions where each fraction requires scheduling. These problems commonly arise in many domains including image processing [1], databases [2], linear algebra [3], and multimedia broadcasting [4].

Scheduling divisible loads is the subject of Divisible Load Scheduling (DLS) theory which was extensively studied in [5] where several network architectures (e.g., linear, bus, tree), task arrangements, and optimality conditions were explored. The underlying assumption of DLS theory is that the processors are obedient, i.e., under no circumstances will the processors “cheat”. In the real world, the assumption is unrealistic as the processors may be owned by autonomous, self-interested entities that have no a priori motivation for cooperation and they are tempted to manipulate the scheduling algorithms in hope of increased benefits. In this type of environment, the processors are properly modeled as strategic agents. New protocols for DLS must account for this self-interested behavior. Mechanism design theory [6] — a field of economics that has recently garnered interest in computer science — provides the framework for solving such problems involving self-interested parties. The theory addresses incentive compatibility: rational agents (self-interested, utility-maximizing) are provided incentives which induce a behavior that maximizes the social welfare. Of interest are the strategyproof mechanisms. Each participant in a mechanism is characterized by private parameters. A strategyproof mechanism will result in a participant maximizing its utility if it truthfully reports its private parameters. Unfortunately, strategyproofness is difficult to obtain in a distributed mechanism as the agents, information, and algorithm are distributed. In this work, we consider the autonomous node model [7] of distributed mechanisms in which the agents have control over both the inputs to the algorithm and the algorithm itself. The self-interested processors will implement a different algorithm if there exist incentives to do so. In designing a distributed mechanism, a suggested strategy, a set of action sequences, one for each agent, is defined. The suggested strategy implements the mechanism’s distributed algorithm. As a design objective, the suggested strategy should be an ex post Nash equilibrium, an equilibrium concept in which an agent will experience a reduction in its utility if it chooses to deviate from the suggested strategy. Having the suggested strategy as an ex post Nash equilibrium ensures that the rational agents choose to abide by it.
In our previous work [8, 9], we showed how DLS theory can be augmented with incentives. We designed a strategyproof mechanism for scheduling divisible loads in bus networks. The mechanism provides incentives to the processors to participate and to report their processing capacities to the centralized, trusted scheduler. The agents maximize their welfare by truthfully reporting their values to the mechanism and executing their assignments as reported.

In this paper, we augment DLS theory with incentives for tree networks, where the network comprises strategic processors. In a tree network, the load is distributed from the root of the tree downward through intermediate processors until all processors are assigned load. We propose the DLS-T (Divisible Load Scheduling - Tree) mechanism to optimally distribute load among the processors in a tree network. The mechanism provides incentives so that the agents correctly execute the two phase algorithm to compute the optimal load allocations. Payments are issued after the agents compute their assignments to ensure that the agents actually processed the load as they reported. Furthermore, we propose the use of a solution bonus to incentivize strongly indifferent agents to follow the mechanism. An agent is strongly indifferent if it implements any strategy that is welfare maximizing and not just the suggested strategy. Thus, the suggested strategy should be the only welfare maximizing strategy, which the solution bonus provides. We show that the suggested strategy of the DLS-T mechanism computes the optimal allocation as an ex post Nash equilibrium. This is important as agents will choose not to deviate from the suggested strategy.

1.1. Related Work

Since its origins [10], the divisible load scheduling problem has been studied extensively resulting in a cohesive theory called Divisible Load Scheduling (DLS) theory. A reference on DLS is [5]. Two recent surveys on DLS are [11] and [12]. In recent years, the investigation into DLS includes new static network topologies [13], arbitrary network topologies [14], and adaptive approaches to load balancing when processor and link capacities are uncertain [15]. Further research has answered questions in multiround installment algorithms [16, 17], linear networks [18], and return messages [19]. This theory has been used for scheduling loads on heterogeneous distributed systems.
in the context of different applications such as image processing [1], databases [2], linear algebra [3], multimedia broadcasting [4], and video-on-demand distribution [20]. Scheduling divisible loads in grids has been investigated in [21]. New results and open research problems in DLS are presented in [13]. All these works assumed that the participants in the load scheduling algorithms are obedient and follow the algorithm. Recently, several researchers considered the mechanism design theory to solve several computational problems that involve self-interested participants. These problems include resource allocation and task scheduling [22, 23, 24], routing [25] and multicast transmission [26]. In their seminal paper, Nisan and Ronen [27] considered for the first time the mechanism design problem in a computational setting. They proposed and studied a VCG (Vickrey-Clarke-Groves) type mechanism for the shortest path in graphs where edges belong to self-interested agents. They also provided a mechanism for solving the task scheduling on unrelated machines problem. A general framework for designing strategyproof mechanisms for one parameter agents was proposed by Archer and Tardos [28]. They developed a general method to design strategyproof mechanisms for optimization problems that have general objective functions and restricted form for valuations. Andelman and Mansour [29] relaxed the conditions for truthfulness of the framework. Mechanisms with verification were examined in [30]. Strategyproof mechanisms with verification combining incentives and DLS were proposed by Grosu and Carroll for bus networks [8, 9, 31] and linear networks [32]. The results and the challenges of designing distributed mechanisms are surveyed in [33]. Mitchell and Teague [7] extended the distributed mechanism in [26] devising a new model where the agents themselves implement the mechanism, thus allowing them to deviate from the algorithm. The authors of [34] defined the concept of faithfulness for distributed mechanisms. A mechanism is faithful if the agents obey the distributed mechanism specification.

### 1.2. Contributions

DLS theory assumes that the participants behave obediently and that they will adhere to the prescribed scheduling algorithms regardless of their individual welfare. In this work, we consider that each processor is owned and operated by an autonomous,
self-interested organization that will deviate from the scheduling algorithms if it is beneficial to do so. We design a distributed mechanism for scheduling divisible loads in a tree network of processors assuming a linear cost model for the processors. We prove that the distributed mechanism computes the optimal allocation as an ex post Nash equilibrium. We devise an incentive scheme to ensure that selfish-and-annoying agents, agents that are strongly indifferent, choose to follow the distributed mechanism specification. Finally, we investigate experimentally the mechanism under different tree network configurations and processor parameters.

1.3. Organization

The paper is structured as follows. In Section 2, we present a description of the divisible load scheduling problem in the context of tree networks. In Section 3, we discuss the mechanism design foundations. In Section 4, we present our proposed mechanism. In Section 5, we prove the mechanism’s properties. In Section 6, we simulate the mechanism to demonstrate its properties. Finally, in Section 7, we draw conclusions and present future directions.

2. Divisible Load Scheduling Problem

In this section, we define the divisible load scheduling problem in tree networks. Much of what we present here are standard DLS results adapted from [5] and [13]. We first consider a distributed system comprising $m + 1$ processors interconnected in a single-level tree network (also called a star network). The single-level tree, $\mathcal{T}$, is composed of a set of processors, $(P_0, P_1, P_2, \ldots, P_m)$, and a set of bidirectional communication links, $(\ell_1, \ell_2, \ldots, \ell_m)$. Processor $P_0$ is the root of $\mathcal{T}$ and it is connected to the terminal processor $P_j$ ($j = 1, 2, \ldots, m$) by link $\ell_j$. Processor $P_i$ ($i = 0, 1, \ldots, m$) is characterized by $t_i$, the time it takes to process a unit load. In the following we refer to $t_i$ as $P_i$’s unit-load processing time. Processor $P_i$ is assigned a fraction $\alpha_i$ of the total load that requires processing by the system. $P_i$ processes its load assignment in time $\alpha_i t_i$. This corresponds to a linear cost model. The parent $P_0$ is the load-originating processor that is responsible for distributing the load to the children. We assume that
the processors have front-ends that allow simultaneous communication and processing and that the root processor can communicate with only one terminal processor at any instant (i.e., we assume the one-port model). Processor $P_0$ transmits a fraction $\alpha_j$ of the total load to child $P_j$ in time $\alpha_j z_j$, where $z_j$ is the time it takes to communicate a unit load from $P_0$ to child $P_j$. We denote by $\alpha = (\alpha_0, \alpha_1, \ldots, \alpha_m)$ the vector of load allocations and by $z = (z_1, z_2, \ldots, z_m)$ the vector of link capacities. Processor $P_i$ finishes its assignment in time $T_i(\alpha)$, which is the total time to receive (if $i \neq 0$) and process its assignment. The execution on a single-level tree is depicted in Figure 1.

The scheduling problem denoted as SINGLE-LEVEL TREE is to determine the optimal load allocation $\alpha$ which minimizes the total execution time

$$T(\alpha) = \max(T_0(\alpha), T_1(\alpha), \ldots, T_m(\alpha)).$$

The finish time $T_i(\alpha)$ of processor $P_i$ is

$$T_i(\alpha) = \begin{cases} 
0 & \text{if } \alpha_i = 0 \\
\alpha_0 t_0 & \text{if } i = 0 \text{ and } \alpha_i > 0 \\
\sum_{j=1}^{i} \alpha_j z_j + \alpha_i t_i & \text{if } i \neq 0 \text{ and } \alpha_i > 0.
\end{cases} \quad (1)$$

The SINGLE-LEVEL TREE scheduling problem can be formalized as

$$\min_{\alpha} T(\alpha) \quad (2)$$

subject to the constraints:

(i) $\alpha_i \geq 0, i = 0, 1, \ldots, m$, and
Algorithm 1 alloc(t, z)

input: single-level tree network with root P₀
and children ordered s.t. \( z_1 \leq z_2 \leq \cdots \leq z_m \)
with unit-load processing times \( t = (t_0, t_1, \ldots, t_m) \)

output: Load allocation \( \alpha \):

1: for \( i \leftarrow 1, m \) do
2: \( k_i \leftarrow \frac{t_i - 1}{t_i} \)
3: end for
4: \( \alpha_0 \leftarrow \frac{1}{1 + \sum_{i=1}^{m} k_i} \)
5: for \( i \leftarrow 1, m \) do
6: \( \alpha_i \leftarrow k_i \alpha_{i-1} \)
7: end for

(ii) \( \sum_{i=0}^{m} \alpha_i = 1 \).

The constraints specify that the load is normalized to one unit of load and that each \( P_i \) receives a fraction \( \alpha_i \in [0, 1] \) of this load. Furthermore, the load is assigned such that no two processors process the same load. An optimal solution to the SINGLE-LEVEL TREE scheduling problem is obtained when \( z_1 \leq z_2 \leq \cdots \leq z_m \) and all processors participate and finish executing their assigned load at the same time [5].

Algorithm 1 determines the optimal allocation for the SINGLE-LEVEL TREE problem [5]. The algorithm is executed by \( P_0 \) when a new load needs processing. In order to compute \( \alpha \), processor \( P_i \) (\( i = 1, 2, \ldots, m \)) reports \( t_i \) to \( P_0 \). It is assumed that \( z_1, z_2, \ldots, z_m \) are known to \( P_0 \).

We can reduce the single-level tree to a single processor having a unit-load processing time equivalent to that of the tree. Figure 2 illustrates the reduction.
Algorithm 2 equiv(α, t, z)

input: single-level tree network with root \( P_0 \)
with allocation \( \alpha = (\alpha_0, \alpha_1, \ldots, \alpha_m) \),
unit-load processing times \( t = (t_0, t_1, \ldots, t_m) \)
and link capacities \( z = (z_1, z_2, \ldots, z_m) \)

output: equivalent unit-load processing time \( t^{eq} \)

1: \( T_0 \leftarrow \alpha_0 t_0 \)
2: \( comm \leftarrow 0 \)
3: for \( i \leftarrow 1, m \) do
4: \( comm \leftarrow comm + \alpha_i z_i \)
5: \( T_i \leftarrow comm + \alpha_i t_i \)
6: end for
7: \( t^{eq} \leftarrow \max\{T_i| i = 0, 1, \ldots, m\} \)

The equivalent unit-load processing time, \( t^{eq} \), by

\[
t^{eq} = \max(T_0(\alpha), T_1(\alpha), \ldots, T_m(\alpha)).
\]

Since \( \sum_{j=0}^{m} \alpha_j = 1 \), the equivalent unit-load processing time of the tree is equal to the finish time of the slowest processor. If \( \alpha \) is optimal then \( t^{eq} \) reduces to

\[
t^{eq} = T_i(\alpha), \quad \forall i = 0, 1, \ldots, m.
\]

Algorithm 2 computes \( t^{eq} \) for any allocation \( \alpha \), not only for the optimal one. If we know that the allocation \( \alpha \) is optimal, the computation of \( t^{eq} \) can be reduced to \( \alpha_0 t_0 \).

We now consider the problem of scheduling divisible loads in multi-level tree networks [13]. Figure 3 depicts an execution on a multi-level tree distributed system composed of eight processors. Notice that after non-terminal processors receive load, they simultaneously begin computing a portion of it and transmitting the remainder to its children. A tree network comprises processor set \( (P_0, P_1, \ldots, P_m) \) (with unit-load processing times \( t_0, t_1, \ldots, t_m \), and link capacities \( z_1, z_2, \ldots, z_m \)), where \( P_0 \) is the root of the tree. We denote by \( T_i \) the single-level subtree with root \( P_i \). We denote by \text{children}(P_i) the set of direct descendants of \( P_i \) and by \text{parent}(P_i), \( P_i \)'s direct predecessor. The load
allocation for $T_i$ is denoted by $\alpha_i = (\alpha_i^e, [\alpha_i^c])$, where $\alpha_i^e$ is the load allocated to $P_i$ and $[\alpha_i^c]_i = (\alpha_i^c | P_i \in \text{children}(P_i))$ is the subvector of load allocations corresponding to the children of $P_i$. Tree $T_i$ has equivalent unit-load processing time $t_i^{eq}$.

In a multi-level tree, the load is distributed from top to bottom, passing through each level. The TREE scheduling problem is defined similarly as SINGLE-LEVEL TREE problem where we desire to optimally distribute load across the multi-level tree such that we minimize the total execution time. The optimal solution is obtained by traversing tree $T$ from bottom to top, replacing single-level subtrees with single equivalent processors until $T$ is reduced to one processor. For single-level subtree $T_i$, we compute load distribution $\alpha_i$ and $t_i^{eq}$ using Algorithm 1 and Algorithm 2 and replace $T_i$ with a processor having equivalent unit-load processing time $t_i^{eq}$. The optimal solution for the TREE scheduling problem is obtained when all processors participate and they all finish executing their assigned load at the same time [13].

Algorithm 3 recursively computes the optimal allocation for the TREE scheduling problem. Starting at the bottom of the tree, it reduces processors with only leaves as children. This process continues until the entire tree is reduced to a single processor.

Once the load allocations for all subtrees are computed we can determine the fraction $\gamma$ of the total load in the system received by processor $P_i$ from its parent as $\gamma = 1$ if $i = 0$, and $\gamma = \alpha_i^e \gamma_p$ if $i > 0$, where $P_p = \text{parent}(P_i)$. We illustrate the computation of $\gamma$ on the example given in Figure 3. $P_0$ is the only processor with initial access to the load and, thus, $\gamma_0 = 1$. $P_0$ determines the fractions of load that are sent to $P_1$ and
Algorithm 3 TREE-SCHED

input: tree $T$
output: allocations for interior processors $\alpha_0, \alpha_1, \ldots$

1: while Reduced tree $T$ has more than one processor do
2: Find a $P_i$ who has children without descendants
3: Compute $\alpha_i$ and $t_{eq}^i$ for $T_i$ using Alg. 1 and Alg. 2
4: Replace $P_i$ and descendants with a single processor
   with unit-load processing time $t_{eq}^i$
5: end while

$P_2$, which are $\gamma_1 = \alpha_0^1 \gamma_0$ and $\gamma_2 = \alpha_0^2 \gamma_0$, respectively. $P_0$ processes itself the remaining load, which is $\alpha_0^0 \gamma_0$. Each non-terminal processor computes the fractions of load to be sent to its children in a similar fashion.

In the above algorithm it is assumed that $P_i$ reports its true unit-load processing time to its parent. When the processors are owned and operated by disparate, autonomous organizations that are self-interested and welfare-maximizing, they will misreport their processing capacities or deviate from the algorithm in hope of generating increased profits. If the processors misreport their unit-load processing times the algorithms presented above will compute the optimal allocation according to the misreported values. This allocation is not the “true” optimal allocation that could have been obtained if the processors reported their true unit-load processing times. In order to obtain the true optimal allocation we need to provide incentives to the processors to report truthfully and follow the scheduling algorithm. In the subsequent sections, we present a mechanism that provides such incentives to the processors.

3. Mechanism Design Framework

In this section, we introduce the main concepts of mechanism design theory [27]. The problem we are considering here is to design a mechanism for scheduling divisible loads. The goal of the mechanism is to determine a feasible load allocation $\alpha$ that gives the minimum total execution time $T(\alpha)$, and to provide incentives to the pro-
cessors (i.e., the agents participating in the mechanism) to report their true unit-load processing times to the mechanism. The mechanism first requests from the processors their unit-load processing times and then computes the load allocation (which is the mechanism’s output) and the payments given to the processors. The load allocation is a vector $\alpha(w) = (\alpha_0(w), \alpha_1(w), \ldots, \alpha_m(w))$, computed according to the agents’ reported unit-load processing times (called bids), $w = (w_0, w_1, \ldots, w_m)$. Here, $w_i$ is the bid of agent $i$.

Each processor (agent) $P_i$ is characterized by three values: (i) $t_i$ - a privately known value representing the true amount of time required to process a unit load (we call this value the true unit-load processing time); (ii) $w_i$ - the reported value of the time required to process a unit load (we call this value the reported unit-load processing time or the bid); and (iii) $\tilde{w}_i$ - a publicly known value representing the actual value of the time to process a unit load (we call this value the actual unit-load processing time).

The preferences of agent $i$ are given by a function called valuation $V_i(\alpha, \tilde{w})$. The valuation $V_i$ is equal to $P_i$’s cost of processing the assigned load (i.e., $V_i = -(P_i$’s processing cost)). A processor $P_i$ wants to maximize its utility $U_i$ which is the sum of its valuation $V_i$ and the payment $Q_i$ given to it. The utility of agent $i$ is $U_i(w, \tilde{w}) = Q_i(w, \tilde{w}) + V_i(\alpha(w), \tilde{w})$, where $Q_i$ is the payment handed by the mechanism to agent $i$ and $\tilde{w}$ is the vector of actual unit-load processing times. The mechanism pays the agents for executing the assigned load. The payments are handed to the agents after the mechanism learns $\tilde{w}$.

$P_i$ bids unit-load processing time $w_i$ to the mechanism; $w_i$ may be different than true unit-load processing time $t_i$. The goal of our mechanism is to select a feasible load allocation $\alpha$ that gives the minimum total execution time $T(\alpha)$. A processor $P_i$ may process its assigned load at a different processing rate given by the actual unit-load processing time $\tilde{w}_i$, where $\tilde{w}_i \geq t_i$. Thus, processor $P_i$ may process its assigned load at a slower rate than its true processing capacity. We cope with this situation by employing a mechanism with verification. The goal of a mechanism with verification is to give incentives to agents such that it is beneficial for them to report their true unit-load processing times and to process their assigned loads using their full processing capacity. A mechanism with verification is defined as follows.
**Definition 3.1** (Mechanism with Verification). A mechanism with verification is characterized by two functions:

(i) The output function \( \alpha(w) = (\alpha_0(w), \alpha_1(w), \ldots, \alpha_m(w)) \). The input to this function is the vector of agents’ bids \( w = (w_0, w_1, \ldots, w_m) \).

(ii) The payment function \( Q(w, \tilde{w}) = (Q_0(w, \tilde{w}), Q_1(w, \tilde{w}), \ldots, Q_m(w, \tilde{w})) \), where \( Q_i(w, \tilde{w}) \) is the payment handed by the mechanism to agent \( i \).

The mechanism designer needs to design an appropriate output function and a payment function that will guarantee that the agents do not have incentives to misreport their parameters. In our case the output function will be based on Algorithm 3.

**Notation.** In the rest of the paper, we denote by \( w_{-i} \) the vector of bids excluding the bid of agent \( i \). The vector \( w \) is represented by \( (w_{-i}, w_i) \).

The following defines an important property of a mechanism, which is, an agent will maximize its utility when \( \tilde{w}_i = w_i = t_i \) regardless of the actions of the other agents.

**Definition 3.2** (Strategyproof Mechanism). A mechanism with verification is called strategyproof if for every agent \( i \) of type \( t_i \) and for all bids \( w_{-i} \) and execution values \( \tilde{w}_{-i} \) of the other agents, the agent’s utility is maximized when it declares its true type \( t_i \) and \( \tilde{w}_i \) is equal to its true type \( t_i \) (i.e., truth-telling and truthful execution is a dominant strategy).

In a distributed mechanism the agents and the algorithms are distributed throughout the network. There are two models for characterizing distributed mechanisms [7]. They differ in the degree of control that the agents have. A mechanism is a tamper-proof mechanism if the agents control the inputs, but not the algorithm. In these types of mechanisms, an agent can only specify its inputs and thus, the only method of cheating is altering its inputs. A more general model is the autonomous node model. A mechanism is an autonomous node mechanism if the agents control both the inputs and the algorithm. The mechanism designer specifies a suggested strategy \( s^*(w) = (s^*_0(w_0), s^*_1(w_1), \ldots, s^*_m(w_m)) \), a vector of strategies with \( s^*_i \) being the sequence of actions for agent \( i \). The suggested strategy implements the mechanism’s algorithm. An agent \( i \) will choose another strategy \( s_i \), where \( s_i \neq s^*_i \) (i.e., another algorithm), if it
is beneficial to do so. We desire a mechanism that will prevent such deviations from the suggested strategy. Such mechanism can be characterized using the ex post Nash equilibrium. In an ex post Nash equilibrium no agent has incentives to deviate from the suggested strategy even if it knows the private information of the other agents.

**Definition 3.3 (Ex Post Nash Equilibrium).** A strategy $s^*$ is an ex post Nash equilibrium if $U_i(s^*(w), \tilde{w}) \geq U_i(s^*_{-i}(w_{-i}), s_i(w_i), \tilde{w})$ for every agent $i$, all strategies $s_i \neq s^*_i$, all bids $w_i$, all values $\tilde{w}$, and all the bids of the other agents, $w_{-i}$.

In the above definition a deviation by a single agent results in a corresponding reduction in its utility. Being rational, the agent will not choose to deviate. The goal of this paper is to design a distributed scheduling mechanism in which the suggested strategy is an ex post Nash equilibrium.

4. The Proposed Mechanism

We propose the DLS-T (Divisible Load Scheduling - Tree) mechanism for scheduling divisible loads in tree networks. DLS-T gives incentives to the processors to follow it and to report their true processing capabilities. The mechanism computes the optimal allocation for a tree network in an ex post Nash equilibrium.

The system model comprises a tree $\mathcal{T}$ of $m + 1$ processors, where the root $P_0$ is obedient and processors $P_1, P_2, \ldots, P_m$ are strategic. Processor $P_0$ is special as it performs functions that ensure the validity of the mechanism. Furthermore, we assume that the links are obedient and that the communication protocols are tamper-proof.

The description of the DLS-T mechanism relies on a few additional assumptions. Informally, we assume the existence of a payment infrastructure and a public key infrastructure (PKI). We assume that all processors have a public cryptographic key set and that the public key from the set is registered with the PKI. Each processor is responsible for protecting the secrecy of its private key. The violation of a private key is the sole responsibility of the owner. Lastly, we assume that any processor can recognize its parent, grandparent, siblings, and children and is capable of verifying their signatures.
Table 1: Notation.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T$</td>
<td>Tree network comprising $m + 1$ processors, $(P_0, P_1, \ldots, P_m)$, and $m$ links, $(\ell_1, \ell_2, \ldots, \ell_m)$.</td>
</tr>
<tr>
<td>$P_0$</td>
<td>The root (load-originating processor) of $T$.</td>
</tr>
<tr>
<td>parent($P_j$)</td>
<td>Parent of $P_j$ ($j = 1, 2, \ldots, m$). Let parent($P_0$) = $P_0$.</td>
</tr>
<tr>
<td>children($P_i$)</td>
<td>Set of $P_i$'s children ($i = 0, 1, \ldots, m$). If $P_i$ is a terminal processor, then children($P_i$) = $\emptyset$.</td>
</tr>
<tr>
<td>$T_i$</td>
<td>Subtree of $T$ with root $P_i$ and all its descendants.</td>
</tr>
<tr>
<td>sig$_i$(msg)</td>
<td>Message msg digitally signed by $P_i$.</td>
</tr>
<tr>
<td>$[x_c]_i$</td>
<td>Subvector $(x_c</td>
</tr>
<tr>
<td>$[x_c]_i^{-k}$</td>
<td>Subvector $(x_c</td>
</tr>
<tr>
<td>$\text{sig}_c(x_c)_i$</td>
<td>Subvector $(\text{sig}_c(x_c)</td>
</tr>
<tr>
<td>$w_i$</td>
<td>$P_i$'s reported unit-load processing time.</td>
</tr>
<tr>
<td>$\tilde{w}_i$</td>
<td>$P_i$'s actual unit-load processing time ($\tilde{w}_i \geq w_i$).</td>
</tr>
<tr>
<td>$w_{eq}$</td>
<td>Equivalent unit-load processing time of $T_i$.</td>
</tr>
<tr>
<td>$z_j$</td>
<td>Link capacity of $\ell_j$, the link connecting $P_j$ to parent($P_j$).</td>
</tr>
<tr>
<td>$w_i$</td>
<td>Vector $(w_i, [w_{eq}]_i)$ of unit-load processing times for reduced (single level) tree $T_i$.</td>
</tr>
<tr>
<td>$w_i^{-j}$</td>
<td>Vector $([w_{eq}]_i \setminus j)$, i.e., $w_i$ without element $w_j$.</td>
</tr>
<tr>
<td>$w_i^{eq}$</td>
<td>Vector $(w_i, [w_{eq}]_i^{-k})$, i.e., $w_i$ without element $w_k$.</td>
</tr>
<tr>
<td>$w_i^\text{sig}$</td>
<td>Signed vector $(\text{sig}_i(w_i), [\text{sig}<em>c(w</em>{eq})]_i)$ of unit-load processing times.</td>
</tr>
<tr>
<td>$\alpha_i$</td>
<td>Vector $(\alpha_i, [\alpha_i]_i)$ of load allocations for the reduced (single level) tree $T_i$.</td>
</tr>
<tr>
<td>$z_i$</td>
<td>Vector $([z_i]_i)$ of the link capacities of the links between $P_i$ and children($P_i$).</td>
</tr>
<tr>
<td>$z_i^{-k}$</td>
<td>Vector $([z_i]_i^{-k})$, i.e., $z_i$ without element $z_k$.</td>
</tr>
<tr>
<td>$\gamma$</td>
<td>Fraction of the total load received by $P_i$ from parent($P_i$).</td>
</tr>
</tbody>
</table>

As we mentioned above, $P_0$ plays the role of an arbitrator. To this end, $P_0$ is connected by a control network to every other processor, allowing direct communication between the processors. Compared to the tree network, this out-of-band network is resource constrained and thus, inferior for transmitting especially large data sets. We further assume that $P_0$ knows the configuration: $P_0$ knows the underlying network graph,
the links, and the link capacities but not the processors’ speeds.

A mechanism without verification (i.e., with payments not determined by the actual unit-load processing times $\tilde{w}_i$) can be used to enforce truthful reporting of processing capacities by the processors. But, such mechanism will not prevent the processors to execute the load at a different processing capacity than the one reported. For this reason, in our setting each processor $P_i$ is augmented with a tamper-proof meter that records $\tilde{w}_i$, the $P_i$’s actual unit-load processing time. If a processor executes the load at a different capacity than the one corresponding to the reported unit-load processing time (it does not need to be the true unit-load processing time) then our mechanism penalizes that processor through payments. Basically, DLS-T enforces two things, truthful reporting (being in an ex-post Nash equilibrium) and truthful execution (by involving payments depending on $\tilde{w}_i$). Here $\tilde{w}_i$ is assumed to be public knowledge and it is being used by the mechanism to compute the payments only after the execution of the load is completed.

The description of the DLS-T mechanism follows. In Tab. 1 we provide the notation used to describe the mechanism.

4.1. DLS-T Mechanism

Phase I (Computing load allocation – bottom-up)

In Phase I, each processor $P_i$, $i = 0, 1, \ldots, m$, computes allocation $\alpha_i$ and equivalent unit-load processing time $w_{eq}^i$ using the distributed protocol given in Alg. 4 and Alg. 5. Distributed Alg. 4 recursively reduces subtrees into equivalent processors beginning at the terminal processors and progressing level by level up the tree until the entire network is reduced to a single equivalent processor. This algorithm is essentially a distributed implementation of Alg. 3.

A nonterminal processor $P_i$ computes only the allocation $\alpha_i = (\alpha^i, [\alpha^c_i])$ for itself ($\alpha^i$) and its children ($[\alpha^c_i]$) using $\text{alloc}(w_i, z_i)$. $P_i$ does not compute the allocation for the whole tree. It also computes the equivalent unit-load processing time $w_{eq}^i$ of its subtree for which it is the root using $\text{equiv}(\alpha_i, w_i, z_i)$. In order to compute $\alpha_i$ and $w_{eq}^i$, $P_i$ needs $w_i = (w_i, [w_{eq}^c_i])$ and $z_i$. $P_i$ knows its own $w_i$ and receives the equivalent unit-load processing times of its children $[w_{eq}^c_i]$ in line 4 of Alg. 4. As mentioned before
Algorithm 4 Computing $\alpha_i$ by processor $P_i$, $i = 0, 1, \ldots, m$

1: if $P_i$ is a terminal processor then
2: \hspace{1em} $w_i^{eq} \leftarrow w_i$
3: else
4: \hspace{1em} Receive $F_c = (\text{sig}_c(w_c^{eq}))$ from $P_c$, $\forall P_c \in \text{children}(P_i)$
5: \hspace{1em} Verify the authenticity of $\text{sig}_c(w_c^{eq})$, $\forall P_c \in \text{children}(P_i)$
6: \hspace{1em} if $\text{sig}_c(w_c^{eq})$ is not authentic then
7: \hspace{1em} \hspace{1em} Terminate the protocol
8: \hspace{1em} end if
9: \hspace{1em} if received $F_c = (\text{sig}_c(w_c^{eq})), F'_c = (\text{sig}_c(w_c^{eq'}))$ then
10: \hspace{1em} \hspace{1em} if $w_c^{eq} \neq w_c^{eq'}$ then
11: \hspace{1em} \hspace{1em} \hspace{1em} if $P_i \neq P_0$ then
12: \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} Send $F_c, F'_c$ to $P_0$
13: \hspace{1em} \hspace{1em} \hspace{1em} end if
14: \hspace{1em} \hspace{1em} end if
15: \hspace{1em} end if
16: end if
17: $w_i \leftarrow (w_i, [w_i^{eq}]), z_i \leftarrow ([z_i])$
18: $\alpha_i \leftarrow \text{alloc}(w_i, z_i), w_i^{eq} \leftarrow \text{equiv}(\alpha_i, w_i, z_i)$
19: end if
20: if $P_i \neq P_0$ then
21: \hspace{1em} Send $F_i = (\text{sig}_i(w_i^{eq}))$ to parent($P_i$)
22: end if

the link capacities $z_i$ are known to $P_i$. Processor $P_i$ may choose to transmit a value $w_i^{eq}$ different from the true unit-load processing time if it benefits it to do so. In the next section, we show that a rational $P_i$ will choose not to do this due to the incentives provided by the mechanism.

If $P_i$ receives unverifiable messages, it immediately terminates the protocol. A message is unverifiable if the signature does not match the content of the message; this is a result of another party attempting to forge the signature. Since the sender cannot be determined, fines cannot be imposed. If it receives contradictory messages $F_c = (\text{sig}_c(w_c^{eq}))$ and $F'_c = (\text{sig}_c(w_c^{eq'}))$, where $w_c^{eq} \neq w_c^{eq'}$, it invokes the resolution
**Algorithm 5** Phase I exception handling by $P_0$

1: if $P_i \neq P_0$ then
2: Receive $F_c = (\text{sig}_c(w_{eq}^c))$, $F'_c = (\text{sig}_c(w_{eq}'^c))$ from $P_i$
3: if $F_c, F'_c$ are authentic and $w_{eq}^c \neq w_{eq}'^c$ then
4: Impose penalty of $C$ on $P_i$ and reward $C$ to $P_c$
5: else
6: Impose penalty of $C$ on $P_i$ and reward $C$ to $P_c$
7: end if
8: end if
9: Terminate the protocol

algorithm given in Alg. 5. The only means to receive authentic and contradictory messages is if the private key $PK_c$ is exposed. If $P_0$ determines that the messages are authentic and contradictory, it imposes a fine $C$ on $P_c$ and gives the amount to $P_i$ as a reward. Otherwise, $P_i$ is fined $C$ and $P_c$ receives $C$. The fine $C$ is chosen such that it is much greater than any benefit a processor can obtain by cheating.

**Phase II (Computing load allocation – top-down)**

In this phase each processor $P_i$, $i = 0, 1, \ldots, m$, computes the fractions of load for its children $\gamma_c$, using Alg. 6 and Alg. 7. The protocol begins at the root and progresses down the tree.

We undo the reductions performed in the previous phase and in the process, we compute $\gamma_i$ for every $P_i$. By definition, $\gamma_0 = 1$ as the root is the only processor who initially has access to the load. $P_i$ computes the fractions of load $\gamma_c$ of its children in line 17 of Alg. 6. To compute these fractions of load $P_i$ needs $\gamma_i$ (sent by its parent in line 6) and the load allocations $\alpha_i^c$ computed by itself in Phase I of the mechanism.

To guarantee that all processors follow the specified algorithm the mechanism requires them to perform additional verification. This verification is not done to determine if the reported unit-load processing times are equal to the true ones but to check that whatever was reported is consistent across the entire execution of the mechanism. Enforcing the correct reporting of unit-load processing times by the processors...
Algorithm 6 Computing $\gamma_i$ by $P_i$, $i = 0, 1, \ldots, m$

1: if $P_i = P_0$ then
2: \[ \gamma_0 \leftarrow 1, \gamma_i \leftarrow \alpha^0_c, \forall P_c \in \text{children}(P_i) \]
3: Send $G_c = (\text{sig}_0(\gamma_0), \text{sig}_0(\gamma_i), \text{sig}_0(\text{sig}_0(w_{0}^{eq}))),$
\[ \text{sig}_0(\text{sig}_c(w_{c}^{eq})), \text{sig}_0(w_{c}^{\text{sig}}), \text{sig}_0(P_0, z_0) \]
to $P_c$, $\forall P_c \in \text{children}(P_i)$
4: else
5: $P_p \leftarrow \text{parent}(P_i), P_{gp} \leftarrow \text{parent}(P_p)$
6: Receive $G_i = (\text{sig}_p(\text{sig}_g_{gp}(\gamma_p)), \text{sig}_p(\gamma_i), \text{sig}_g_{gp}(\text{sig}_p(w_{gp}^{eq}))),$
\[ \text{sig}_g_{gp}(\text{sig}_s(w_{s}^{eq})), \text{sig}_s(w_{s}^{\text{sig}}), \text{sig}_0(P_p, z_p) \]
from $P_p$
7: if any signature in $G_i$ is not verifiable then
8: Terminate the protocol
9: end if
10: if $\text{sig}_0(P_j, z_j) \in G_i$ and $P_j \neq P_p$ then
11: Send $G_i$ to $P_0$ and invoke Alg. 7
12: end if
13: $\alpha_p \leftarrow \text{alloc}(w_{p}, z_p), w_{p}^{eq} \leftarrow \text{equiv}(\alpha_p, w_{p}, z_p)$
14: if $w_{p}^{eq} \neq w_{p}\alpha_p^p$ or $\gamma \neq \alpha^p_P$ then
15: Send $G_i$ to $P_0$ and invoke Alg. 7
16: end if
17: $\gamma_i \leftarrow \alpha^0_c \gamma_i, \forall P_c \in \text{children}(P_i)$
18: Send $G_c = (\text{sig}_c(\text{sig}_p(\gamma_i)), \text{sig}_c(\gamma_i), \text{sig}_p(\text{sig}_c(w_{c}^{eq}))),$
\[ \text{sig}_c(\text{sig}_s(w_{s}^{eq})), \text{sig}_s(w_{s}^{\text{sig}}), \text{sig}_0(P_i, z_i) \]
to $P_c$, $\forall P_c \in \text{children}(P_i)$
19: end if

will be done through payments in Phase IV. We now describe the additional verification performed in Phase II. Let $P_p = \text{parent}(P_i)$ and $P_{gp} = \text{parent}(P_p)$. Processor $P_i$ receives message $G_i = (\text{sig}_p(\text{sig}_g_{gp}(\gamma_p)), \text{sig}_p(\gamma_i), \text{sig}_g_{gp}(\text{sig}_p(w_{gp}^{eq}))),$ $\text{sig}_g_{gp}(\text{sig}_s(w_{s}^{eq})), \text{sig}_s(w_{s}^{\text{sig}}), \text{sig}_0(P_p, z_p)$ from $P_p$ and verifies the authenticity of the signatures within the message. As we assume that $P_0$ knows the configuration of the network, each $P_i$ has obtained a vector of its link capacities signed by $P_0, \text{sig}_0(P_i, z_i)$. The vector contains the identifier $P_i$ to thwart processors from exchanging vectors. Note that within $G_i$ several
Algorithm 7 Phase II exception handling by $P_0$

1: Receive $G_i$ from $P_i$
2: $P_p \leftarrow \text{parent}(P_i)$
3: if any signature in $G_i$ is not verifiable then
4: $\text{FinePp} \leftarrow \text{false}$
5: else if $\text{sig}_0(P_j, z_j) \in G_i$ and $P_j \neq P_p$ then
6: $\text{FinePp} \leftarrow \text{true}$
7: else
8: $\alpha_p \leftarrow \text{alloc}(w_p, z_p)$
9: if $w_p^{eq} \neq \alpha_p^p w_p$ or $\gamma_i \neq \gamma_p \alpha_p^p$ then
10: $\text{FinePp} \leftarrow \text{true}$
11: else
12: $\text{FinePp} \leftarrow \text{false}$
13: end if
14: end if
15: if $\text{FinePp} = \text{true}$ then
16: Impose penalty of $C$ on $P_p$ and reward $C$ to $P_i$
17: else
18: Impose penalty of $C$ on $P_i$ and reward $C$ to $P_p$
19: end if
20: Terminate the protocol

values have chained signatures. Load fraction $\gamma_p$ is computed by $P_{gp} = \text{parent}(P_p)$ and is signed by it. Processor $P_{gp}$ sends $\text{sig}_{gp}(\gamma_p)$ to $P_p$, which then signs the signed value and sends it to $P_i$. A similar signing process applies to $\gamma_i$. In Phase I, $P_p$ sends $\text{sig}_{gp}(w_p^{eq})$ to $P_{gp}$. Processor $P_{gp}$ signs it and returns $\text{sig}_{gp}(\text{sig}_{gp}(w_p^{eq}))$ to $P_p$ in the second phase. Similar reasoning applies to $\text{sig}_{gp}(\text{sig}_{i}(w_i^{eq}))$.

Processor $P_i$ verifies that $\text{alloc}(w_p, z_p)$ gives $w_p^{eq}$ and that $\gamma_i = \alpha_i^p \gamma_p$. $P_i$ has all the information necessary to verify these since $w_p, z_p, w_p^{eq}, \gamma_i$ and $\gamma_p$ are received by $P_i$ in line 6 of Alg. 6. If the verified values do not agree, $P_i$ invokes Alg. 7.

Processor $P_0$ executes Alg. 7. Processor $P_0$ receives $G_i$ from $P_i$ and verifies its authenticity. If the message is authentic, it then verifies the correctness of the values
contained within $G_i$. If one or more values are incorrect, $P_0$ charges $P_p$ a fine $C$, which is then awarded to $P_i$. Otherwise, $P_0$ penalizes $P_i$ a sum $C$ and awards it to $P_p$.

A processor $P_i$ invokes Alg. 7 when some anomaly is detected. It does it by choice as if it forgoes tattling, another processor will choose to tattle, resulting in $P_i$ being penalized in later phases of the protocol.

**Phase III (Load distribution and execution)**

Phase III of the mechanism consists of Alg. 8 and Alg. 9. In this phase, each processor receives its assignment and it concurrently computes and distributes load to its descendants. The protocol progresses from root down the tree.

Processor $P_i, i = 1, 2, \ldots, m$, receives a fraction $\gamma_i$ of load from its parent, $\text{parent}(P_i)$. Allocation $\gamma_i$ may not equal $\gamma_i$ (the value computed in Phase II) as predecessors may under-allocate themselves, passing additional load to their children. If $P_p$ is the predecessor who under-allocated itself, then $\alpha_i^p$, the true amount it allocated itself, is $\tilde{\alpha}_i^p \neq \alpha_i^p$. At this point $P_i$, the child of $P_p$, must accept the additional load. Once the load is received in its entirety, $P_i$ begins processing its load $\gamma_i\alpha_i^c$ and distributes $\gamma_i\alpha_i^c$.

**Algorithm 8 Load distrib. & execution by $P_i, i = 0, 1, \ldots, m$**

```plaintext
1: if $P_i = P_0$ then
2:   Process its load $\alpha_0^0$ at the capacity corresponding to $t_0 = w_0 = \tilde{w}_0$.
   while concurrently sending load $\gamma_c$ to $P_c, \forall P_c \in \text{children}(P_0)$
3: else
4:   Receive load $\gamma_i$ from parent($P_i$)
5:   Process load $\gamma_i\alpha_i^c$ at a rate corresponding to $\tilde{w}_i$.
   while concurrently sending load $\gamma_c$ to $P_c, \forall P_c \in \text{children}(P_i)$
6: if $\gamma_i \neq \gamma_i$ then
7:   Send $G_i, L_i$ to $P_0$
8:   Invoke Alg. 9
9: end if
10: end if
```
Algorithm 9 Phase III exception handling by $P_0$

1: Receive $G_i, L_i$ from $P_i$
2: Compute $\tilde{\gamma}_i$ from $L_i$
3: if all signatures in $G_i$ are verifiable and $\tilde{\gamma}_i > \gamma_i$ then
4: $P_p \leftarrow \text{parent}(P_i), P_c \leftarrow P_i$
5: while $P_p \neq P_0$ do
6: Request $G_p$ from $P_p$
7: if $\tilde{\gamma}_p > \alpha_p \tilde{\gamma}_p$ then
8: $P_p$ is fined $C$
9: end if
10: $P_c \leftarrow P_p, P_p \leftarrow \text{parent}(P_c)$
11: end while
12: Award the pooled fine to $P_i$
13: else
14: Fine $C$ to $P_i$
15: end if
16: Terminate the protocol

To detect overloaded processors, we assume that the data received by $P_i$ is embedded with a device $L_i$ which permits the detection of $\tilde{\gamma}_i > \gamma_i$. If determined to be true, $P_i$ invokes the conflict resolution protocol given in Alg. 9. Processor $P_0$ traverses the predecessors of $P_i$ fining all processors that over-allocated their children. The fines are pooled and then rewarded to $P_i$. A processor does not obtain any benefit by over-allocating itself since by doing so its processing cost will increase leading to a reduction of its utility.

Phase IV (Payment computation)

An example of a simple device $L_c$ would be preparing the data by dividing it into equal-sized blocks and appending a unique, random identifier to each block. The identifier space must be large enough so that the probability of an agent successfully guessing a valid identifier is small. Submitting the identifiers would permit $P_c$ to show the amount of data it received.
Algorithm 10 Computing payment \( Q_i \) by \( P_i, i = 1, 2, \ldots, m \)

1: Compute payment \( Q_i \) using (7)

2: Send \( Q_i \) to \( P_0 \)

3: if selected by \( P_0 \) to be verified then

4: Send \( \text{Proof}_i = (G_i, \text{sig}_0(\bar{w}_i), \text{sig}_i(\bar{y}_i), L_i, \text{sig}_0(P_i, p_i)) \)

5: end if

Each processor computes its payment. This phase consists of Alg. 10 and Alg. 11. Each processor \( P_i, i = 1, 2, \ldots, m \), executes Alg. 10 to compute its own payment. In the following, \( \bar{\alpha}_i \) is the actual load allocation for \( P_i \); \( \bar{w}_i \) is the actual unit-load processing time of \( P_i \); and \( \bar{y}_i \) is the actual fraction of total work received by \( P_i \). Let \( P_p = \text{parent}(P_i) \).

The goal of processor \( P_i \) is to maximize its utility. The utility of \( P_i \) is

\[
U_i(\bar{w}_i, \bar{z}_i, \bar{w}_p, \bar{z}_p, \bar{\alpha}_i, \bar{\gamma}_i) = V_i(\bar{\alpha}_i, \bar{\gamma}_i) + Q_i(\bar{w}_i, \bar{z}_i, \bar{w}_p, \bar{z}_p, \bar{\alpha}_i, \bar{\gamma}_i),
\]

where \( V_i \) is the valuation function, and \( Q_i \) is the payment function. The valuation function \( V_i \) is

\[
V_i(\bar{\alpha}_i, \bar{\gamma}_i) = -\bar{\gamma}_i\bar{\alpha}_i\bar{w}_i\tag{6}
\]

Thus, \( P_i \)'s valuation is equivalent to the cost of processing \( \bar{\gamma}_i\bar{\alpha}_i \) amount of load. In the following we denote by \( \alpha(\bar{w}_i, p_i) \) the allocation given by \( \text{alloc}(\bar{w}_i, p_i) \).

The payment function \( Q_i \) is

\[
Q_i(\bar{w}_i, \bar{z}_i, \bar{w}_p, \bar{z}_p, \bar{\alpha}_i, \bar{\gamma}_i) = \begin{cases} 
0 & \text{if } \bar{\gamma}_i = 0, \\
C_i(\alpha(\bar{w}_i, \bar{z}_i), \bar{\gamma}_i) + B_i(\bar{w}_i, \bar{z}_i, \bar{w}_p, \bar{\alpha}_i, \bar{\gamma}_i) & \text{if } \bar{\gamma}_i > 0,
\end{cases}
\]

where

\[
C_i(\alpha(\bar{w}_i, \bar{z}_i), \bar{\gamma}_i) = \bar{\gamma}_i\bar{\alpha}_i\bar{w}_i\tag{8}
\]

is the compensation function and

\[
B_i(\bar{w}_i, \bar{z}_i, \bar{w}_p, \bar{\alpha}_i, \bar{\gamma}_i) = w_{p_i}^{\alpha}(\alpha(\bar{w}_p, \bar{z}_p), \bar{w}_p, \bar{\alpha}_i, \bar{\gamma}_i) - w_{p_i}^{\alpha}(\alpha(\bar{w}_p, \bar{z}_p), \bar{w}_p, \bar{\alpha}_i, \bar{\gamma}_i, \bar{z}_i, \bar{z}_i, \bar{\gamma}_i).
\]

\[
\tag{9}
\]
Algorithm 11 Verification by $P_0$

1: for $i = 1, 2, \ldots, m$ do
2:   Receive $Q_i$
3:   Request Proof$_i$ from $P_i$ with probability $q$
4:   if Proof$_i$ is requested then
5:     Receive Proof$_i$ from $P_i$
6:     Verify that $Q_i$ can be obtained from Proof$_i$
7:     if $Q_i$ is not correct then
8:       Impose penalty $C/q$ on $P_i$
9:     else
10:       Pay $Q_i$ to $P_i$
11:   end if
12: end if
13: end for

is the work bonus function. The compensation function compensates for the cost of processing the load, while the bonus function gives a bonus that is equal to the contribution of $P_i$’s subtree in reducing the execution time required to process the total load.

The function $w^{eq}_p(\alpha(w^{eq}_p, z^p), w^{eq}_p, z^p)$ is the optimal equivalent unit-load processing time of the single-level subtree (after reduction) with root $P_p$ and without processor $P_i$ and its descendants. Finally, $w^{eq}_p(\alpha(w, z), (w^{eq}_i, w^{eq}(\alpha(w, z), (w^{eq}_i, \tilde{w})), z))$ is the equivalent unit-load processing time of the subtree rooted at $P_p$ adjusted for the actual unit-load processing time of $P_i$. The payment $Q_i$ is specifically designed to give incentives to the processors to report their true parameters and to process the assigned load at their full processing capacities. We will formally prove this property in Section 5.

All the information necessary for computing the payment $Q_i(w_i, z_i, w_p, z_p, \tilde{w}_i, \gamma_i)$ is available at $P_i$ as follows: $[w^{eq}_c]_i$ component of $w_i = (w_i, [w^{eq}_c]_i)$ was received from $P_i$’s children in Phase II; $z_i$ is known to $P_i$ by assumption; $w_p$ and $z_p$ were received by $P_i$ in Phase II; $\tilde{w}_i$ is publicly known; and $\gamma_i$ is known by $P_i$ since it is the fraction of load it received from its parent for processing in Phase III.
Processor $P_i$ saves

$$\text{Proof}_i = (G_i, \text{sig}_0(\tilde{w}_i), \text{sig}_i(\tilde{y}_i), L_i, \text{sig}_0(P_i, z_i))$$  \hspace{1cm} (10)$$

as evidence of correctly computing the payment. It submits bill $Q_i$ to $P_0$. With probability $q$ (where $0 < q \leq 1$), processor $P_0$ requests $\text{Proof}_i$ from $P_i$. Processor $P_0$ verifies that the entries are authentic, that $\tilde{y}_i$ can be derived from $L_i$, and that $Q_i$ is correctly computed. If $P_i$ fails to provide a proof or the proof is invalid, $P_i$ is fined $C/q$. Otherwise, $P_i$ is paid $Q_i$.

This concludes the description of the DLS-T mechanism. The mechanism as described above is valid for selfish-but-agreeable agents but not for selfish-and-annoying agents. Recall that a selfish-but-agreeable agent will deviate from the algorithm only if it strictly improves its utility, while a selfish-and-annoying agent will only follow the algorithm if it is the only action that maximizes its utility. In the case of DLS-T, selfish-and-annoying processors will subvert the mechanism by performing undesirable actions (e.g., corrupting data, sending the same data set to multiple children, etc.) where their behavior is not constrained by incentives or penalties. If the load is associated with a problem where the solution can be verified (e.g., searches, factorizations), we can easily amend the mechanism to tolerate selfish-and-annoying processors. This is achieved by modifying (7) as follows

$$Q_i(w_i, z_i, w_p, z_p, \tilde{w}_i, \tilde{y}_i) =
\begin{cases}
0 & \text{if } \tilde{y}_i = 0, \\
C_i(\mathbf{a}(w_i, z_i), \tilde{w}_i, \tilde{y}_i) + B_i(w_i, z_i, w_p, z_p, \tilde{w}_i) + S & \text{if } \tilde{y}_i > 0,
\end{cases}$$

where $S = 0$, if a solution is not found, and $S = s$, if a solution is found. The function $S$ is called the solution bonus. The bonus $s$ is a small, positive quantity that rewards agents for following the given algorithm. Selfish-and-annoying agents will not risk the loss of $s$; hence, they will not deviate.
5. DLS-T Properties

In this section, we study the properties of the DLS-T mechanism. As mechanism designers we would like every processor to follow the suggested strategy, i.e., the DLS-T mechanism. We prove that the suggested strategy is an ex post Nash equilibrium and, thus, rational agents will not have incentives to deviate from DLS-T. To prove this, we first show that no processor has an incentive to deviate from the algorithm. Then we show that if the processors do not deviate (but they can still execute differently than they bid), then a truthful processor maximizes its utility, regardless of the others’ bids. Combining the two, we prove that the suggested strategy is an ex post Nash equilibrium.

**Lemma 5.1.** A selfish-but-agreeable processor will be fined for deviating from DLS-T.

*Proof.* Let processor \( P_i \) be a selfish-but-agreeable agent and \( P_c \in \text{children}(P_i) \). A selfish-but-agreeable agent will deviate if the action increases its utility. Processor \( P_i \) may deviate from the protocol by either (i) sending contradictory messages, (ii) incorrectly computing \( w_i^q \) or \( \gamma_c \), (iii) decreasing its workload by increasing the workload of its children (i.e., \( \tilde{\gamma}_c > \gamma_c \)), (iv) requesting a payment greater than \( Q_i \), (v) falsely accusing a processor of cheating, or (vi) tampering with \( \text{sig}_0(P_i, z_i) \). Processor \( P_i \) will not deviate in other fashions (e.g., corrupt data) because there is no benefit to do so. To combat deviations, incentives are provided to processors for monitoring one another. In case (i), the recipient will report \( P_i \) and obtain reward \( C \); processor \( P_i \) will be fined \( C \). Fine \( C \) is a deterrent as it is greater than any profit attainable by cheating. In case (ii), child \( P_c \) will verify the computations in Phase II and will report \( P_i \) for reward \( C \) if the computation cannot be validated. Again, \( P_i \) is fined \( C \) for deviating from the algorithm. In case (iii), child \( P_c \) receives a reward of at least \( C \) for reporting \( P_i \) and processor \( P_i \) is penalized \( C \). In case (iv), the fine \( C/q \) (where \( 0 < q \leq 1 \) is the probability of processor \( P_0 \) requesting a proof) is a deterrent for over billing. The complete proof for case (iv) can be found in [7]. In case (v), processor \( P_i \) does not have the evidence to support its claim; thus, \( P_i \) is fined \( C \). Finally, a child of \( P_i \) will not be able to verify the signature of \( \text{sig}_0(P_i, z_i) \) and will report \( P_i \) to \( P_0 \), resulting in fine \( C \) imposed on \( P_i \). \( \square \)
Lemma 5.2. A processor receives a fine only if it has deviated from DLS-T.

Proof. Processor $P_i$ is fined for either deviating from the protocol, or another processor $P_j$ produces contradictory messages signed by $P_i$. In the first case, $P_i$ clearly deviates from DLS-T. In the second case, $P_j$ sends the messages either by successfully forging signatures, or by possessing the private key $SK_i$. We assume that the forging of signatures is impossible. Processor $P_j$ obtains $SK_i$ either by $P_j$ sharing it or by stealing it from $P_i$. It is a violation of the mechanism for a second party to possess $SK_i$. Thus, $P_i$ is fined for protocol deviation.

Theorem 5.1 (Selfish-but-Agreeable Agent Compliance). A selfish-but-agreeable processor does not have incentives to deviate from DLS-T.

Proof. From Lemma 5.1 and 5.2, a selfish-but-agreeable processor will be fined for and only for deviating. Therefore, the processor does not have incentives to deviate from DLS-T.

Theorem 5.2 (Selfish-and-Annoying Agent Compliance). A selfish-and-annoying agent does not have incentives to deviate from DLS-T if the solution bonus function is employed.

Proof. Let processor $P_i$ be a selfish-and-annoying agent. Theorem 5.1 handles the cases in which deviation results in greater utility, i.e., $U'_i - U_i > 0$, where $U'_i$ is the utility of a deviating $P_i$. Processor $P_i$ will also deviate if it does not reduce its utility, i.e., $U'_i - U_i = 0$. Such actions include data corruption and sending the same data to different children. These actions reduce the possibility of obtaining a solution to the given problem and, thus, reduce the possibility of receiving the solution bonus. Processor $P_i$ is welfare maximizing and thus, will not choose to do so. Therefore, processor $P_i$ does not have incentives to deviate from DLS-T.

We now prove that if no deviation occurs, the mechanism guarantees that a processor will obtain the maximum utility only by reporting its true parameters. This result combined with Theorems 5.1 and 5.2 allows us to show that the DLS-T mechanism computes the optimal allocation in an ex-post Nash equilibrium (Theorem 5.3).
Lemma 5.3. Assuming that all the processors follow the DLS-T mechanism, a processor cannot increase its utility by misreporting its unit-load processing time.

Proof. Let \( P_p = \text{parent}(P_i) \). The utility \( U_i \) of processor \( P_i \) (\( i = 1, \ldots, m \)) is

\[
U_i = V_i + Q_i
\]

\[
= -\gamma_i \alpha_i \tilde{w}_i + \gamma_i \alpha_i \tilde{w}_i + w^{eq}_p(\alpha(w^{-i}_p, z^{-i}_p), w^{-i}_p, z^{-i}_p) -
\]

\[
w^{eq}_p(\alpha((w^{-i}_p, w^{eq}_i(\alpha(w^{-i}_p, z_i), (w^{-i}_p, \tilde{w}_i))), z_p),
\]

\[
(w^{-i}_p, w^{eq}_i(\alpha((w^{-i}_p, w_i), (w^{-i}_p, \tilde{w}_i))), z_p)
\]

Since the processors cannot manipulate \( z_p \) and \( z_i \), we simplify the equations in the proof by eliminating them from the parameters of \( w^{eq}_p \). The equation above becomes

\[
U_i = -\gamma_i \alpha_i \tilde{w}_i + \gamma_i \alpha_i \tilde{w}_i + w^{eq}_p(\alpha(w^{-i}_p), w^{-i}_p) -
\]

\[
w^{eq}_p(\alpha(w^{-i}_p, w^{eq}_i(\alpha(w^{-i}_p, w_i), (w^{-i}_p, \tilde{w}_i))),
\]

\[
(w^{-i}_p, w^{eq}_i(\alpha(w^{-i}_p, w_i), (w^{-i}_p, \tilde{w}_i)))
\]

We assume that the processors do not deviate and thus abide by the computed load allocations; therefore, \( \alpha^{i}_i = \alpha^{j}_i \). The utility \( U_i \) is

\[
U_i = w^{eq}_p(\alpha(w^{-i}_p), w^{-i}_p) -
\]

\[
w^{eq}_p(\alpha(w^{-i}_p, w^{eq}_i(\alpha(w^{-i}_p, w_i), (w^{-i}_p, \tilde{w}_i))),
\]

\[
(w^{-i}_p, w^{eq}_i(\alpha(w^{-i}_p, w_i), (w^{-i}_p, \tilde{w}_i)))
\]

We consider two cases:

(i) \( \tilde{w}_i = t_i \), i.e., processor \( P_i \) computes the load at full capacity. If \( P_i \) bids its true value, \( w_{i|e} = t_i \), then its utility denoted as \( U_{i|e} \) is

\[
U_{i|e} = w^{eq}_p(\alpha(w^{-i}_p), w^{-i}_p) -
\]

\[
w^{eq}_p(\alpha((w^{-i}_p, w^{eq}_i(\alpha(w^{-i}_p, t_i), (w^{-i}_p, t_i))),
\]

\[
(w^{-i}_p, w^{eq}_i(\alpha((w^{-i}_p, t_i), (w^{-i}_p, t_i))),
\]

\[
(w^{-i}_p, w^{eq}_p(\alpha(w^{-i}_p, w^{-i}_p), w^{-i}_p)) - w^{eq}_{p|e}.
\]

27
If $P_i$ bids lower ($w_{i[l]} < t_i$), then its utility $U_{i[l]}$ is

$$U_{i[l]} = w^e_{p_i} (\alpha(w_{p_i}^{-l}), w_{p_i}^{-l}) -$$

$$w^e_{p_i} (\alpha((w_{p_i}^{-l}, w_{i[l]}), (w_{-i}^{-l}, t_i))) \leq w^e_{p_i} (\alpha(w_{p_i}^{-l}, w_{i[l]}), (w_{-i}^{-l}, t_i)),$$

$$= w^e_{p_i} (\alpha(w_{p_i}^{-l}), w_{p_i}^{-l}) - w^e_{p_i}.$$

We want to show that $U_{i[e]} \geq U_{i[l]}$, which reduces to showing $w^e_{p_i[e]} \leq w^e_{p_i[l]}$. By the SINGLE-LEVEL TREE algorithm, we know that $\alpha((w_{p_i}^{-l}, t_i))$ is the optimal allocation. By bidding lower than the true value, $P_i$ is assigned more load and the other processors in the subtree rooted at $P_p$ are assigned less load. The greater load increases the execution time of $P_i$ and increases the equivalent unit-load processing time (larger $w^e_{p_i}$ corresponding to slower execution) such that $w^e_{p_i[e]} \leq w^e_{p_i[l]}$. Therefore, $U_{i[e]} \geq U_{i[l]}$.

The other possibility is that $P_i$ bids higher ($w_{i[h]} > t_i$). Its utility $U_{i[h]}$ is

$$U_{i[h]} = w^e_{p_i} (\alpha(w_{p_i}^{-h}), w_{p_i}^{-h}) -$$

$$w^e_{p_i} (\alpha((w_{p_i}^{-h}, w_{i[h]}), (w_{-i}^{-h}, t_i))) \leq w^e_{p_i} (\alpha(w_{p_i}^{-h}, w_{i[h]}), (w_{-i}^{-h}, t_i)),$$

$$= w^e_{p_i} (\alpha(w_{p_i}^{-h}), w_{p_i}^{-h}) - w^e_{p_i}.$$

Similar to above, we want to show $U_{i[e]} \geq U_{i[h]}$. Bidding higher than the true value, results in reduced load to $P_i$ and increased load to the other processors. Since $\alpha((w_{p_i}^{-l}, t_i))$ is optimal, $w^e_{p_i[e]} \leq w^e_{p_i[h]}$ and thus, $U_{i[e]} \geq U_{i[l]}$.

(ii) $\tilde{w}_i > t_i$, i.e., processor $P_i$ computes the load slower than its full processing capacity. A similar argument as in case (i) applies. □

**Theorem 5.3 (Ex Post Equilibrium).** The suggested strategy, which is executing the DLS-T mechanism to compute the optimal allocation, is an ex post Nash equilibrium.

**Proof.** Lemma 5.3 states that if no deviation occurs, the mechanism guarantees that a processor will obtain the maximum utility only by reporting its true parameters. Theorems 5.1 and 5.2 state that processors have no incentives to deviate from the mechanism. Therefore, the default strategy is an ex post Nash equilibrium. □
Another useful property of a mechanism is that a processor will never obtain negative utility (i.e., \( U_i \geq 0 \)) at the equilibrium.

**Theorem 5.4.** At the ex-post Nash equilibrium given by the DLS-T, the agents will never incur a loss (i.e., \( U_i \geq 0 \), \( i = 1, 2, \ldots, m \)).

**Proof.** At the ex-post Nash equilibrium all processors follow the DLS-T mechanism.

The utility of processor \( P_i \) (with parent \( P_p \)) when it bids its true value is given by (12). The equivalent unit-load processing time \( w_p^{eq}(\alpha(w_p^{-i}), w_p^{-i}) \) is obtained by using all the processors except for \( P_i \) and its descendants. We know that the optimal equivalent unit-load processing time \( w_p^{eq} \) is obtained when all processors participate [13] and thus,

\[
 w_p^{eq}(\alpha((w_p^{-i}, t_i)), (w_p^{-i}, w_i^{eq}(\alpha(w_i^{-i}, t_i), (w_i^{-i}, t_i)))) ,
\]

\[
 (w_p^{-i}, w_i^{eq}(\alpha(w_i^{-i}, t_i), (w_i^{-i}, t_i))) \leq w_p^{eq}(\alpha(w_p^{-i}), w_p^{-i}) ,
\]

This implies that \( U_i \geq 0 \). \qed

Now a note on the communication complexity of DLS-T. Disregarding the communication with \( P_0 \) for algorithm enforcement and the distribution of load, we see that Phase II dominates communication in terms of number of messages and message size. In that phase, the root of a single-level subtree must send a \( m+1 \)-sized message to each of its \( m \) children. The worst case is when the tree is a single-level tree with \( n \) processors in which the root must transmit an \( n \)-sized message to each of its \( n-1 \) children; thus, the communication complexity in the worst-case is \( \Theta(n^2) \).

The communication between the processors and the root happens only when an exception is detected (which should occur very rarely due to the incentives provided by the mechanism) and when one of the processors is audited (which happens with probability \( q \)). Since most of the communication in DLS-T takes place only between neighbors (parent and its children) increasing the number of nodes is not going to have a strong impact on the communication. Thus, the mechanism is scalable in terms of communication requirements.
6. Experimental results

We study by simulation the DLS-T distributed scheduling mechanism. The simulation was conducted using software written specifically for modeling the tree topology and the mechanism using the Ruby programming language [35]. We consider three networks, each comprising fifteen processors labeled $P_0, P_1, \ldots, P_{14}$. The first network, which we depict in Figure 4, is a balanced binary tree. The other two networks, Figure 5 and Figure 6, are unbalanced binary trees. In each simulation processor $P_1$ cheats and we measure the resulting impact on the makespan, $P_1$’s utility, and $P_1$’s payment. We choose $P_1$ as the cheating processor since its position gives it great influence over the allocations. We present the true values for the processors in Tab. 2. Note that we consider two types of $P_1$, a “fast” setup with $w_1 = 250$ and a “slow” setup with $w_1 = 500$. In all networks, the communication latency is $z = 1$ which ensures that the systems are computationally bound. In all simulations we consider that a unit of load is assigned for processing to the whole system. We choose to work only with unit loads in order to make the results easier to interpret and understand. The comparison of the fast and slow networks permits us to examine the effects of varying the processor parameters. By simulating the different network types, we examine the influence of the number of $P_1$’s descendants on the mechanism execution. We only consider the effect of misreporting the parameters and the effect of processing the load at different rates than the ones reported. These deviations are handled implicitly by the designed payment function and are the most suitable for a study by simulation. The other possible deviations from the mechanism are handled by using the fines which will lead to huge penalties making the study by simulation unnecessary. Furthermore, we only consider systems with a single cheating processor. We showed that DLS-T admits an ex post Nash equilibrium (Thm. 5.3). Because of this, a processor will maximize its utility by bidding and executing truthfully, regardless of the actions chosen by the others. Consequently, the study of systems with multiple non-colluding cheating processors is not conducted as they will exhibit effects similar to those of systems with a single cheating processor.

For each network we examine eight cases:
Figure 4: Balanced binary tree network model.

Figure 5: Unbalanced binary tree ($P_1$ has more descendants than $P_2$).

Figure 6: Unbalanced binary tree ($P_2$ has more descendants than $P_1$).

(1) $t_1 = w_1 = \tilde{w}_1$ ($P_1$ bids truthfully and processes the load as declared);

(2) $t_1 = w_1 < \tilde{w}_1$ ($P_1$ bids truthfully but processes the load at a slower rate);

(3) $t_1 < w_1 = \tilde{w}_1$ ($P_1$ bids a higher unit-load processing time than its true one and it processes the load as declared);

(4) $t_1 = \tilde{w}_1 < w_1$ ($P_1$ bids a higher unit-load processing time than its true one but processes the load at its true processing capacity);

(5) $t_1 < \tilde{w}_1 < w_1$ ($P_1$ bids a unit-load processing time higher than the actual one and processes the load at a rate slower than the true one);

(6) $t_1 < w_1 < \tilde{w}_1$ ($P_1$ bids a unit-load processing time higher than its true one and processes the load at a rate slower than it declared);
The processors’ true unit-load processing times are given in Table 2:

<table>
<thead>
<tr>
<th>$t_0$</th>
<th>$t_1$</th>
<th>$t_2$–$t_6$</th>
<th>$t_7$–$t_{14}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>fast</td>
<td>slow</td>
<td>500</td>
</tr>
</tbody>
</table>

Table 3: Fast $P_1$’s bids, $w_1$, and actual unit-load processing time, $\tilde{w}_1$.

<table>
<thead>
<tr>
<th>Case</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>$w_1$</td>
<td>250</td>
<td>250</td>
<td>750</td>
<td>750</td>
<td>750</td>
<td>500</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>$\tilde{w}_1$</td>
<td>250</td>
<td>500</td>
<td>750</td>
<td>250</td>
<td>500</td>
<td>750</td>
<td>250</td>
<td>500</td>
</tr>
</tbody>
</table>

Table 4: Slow $P_1$’s bids, $w_1$, and actual unit-load processing time, $\tilde{w}_1$.

<table>
<thead>
<tr>
<th>Case</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>$w_1$</td>
<td>500</td>
<td>500</td>
<td>750</td>
<td>750</td>
<td>1000</td>
<td>750</td>
<td>250</td>
<td>250</td>
</tr>
<tr>
<td>$\tilde{w}_1$</td>
<td>500</td>
<td>750</td>
<td>750</td>
<td>500</td>
<td>750</td>
<td>1000</td>
<td>500</td>
<td>750</td>
</tr>
</tbody>
</table>

(7) $t_1 = \tilde{w}_1 > w_1$ ($P_1$ bids a unit-load processing time lower than its true one and processes the load at its true processing rate);

(8) $w_1 < t_1 < \tilde{w}_1$ ($P_1$ bids a unit-load processing time lower than its true one and processes the load at a rate slower than the true one);

The bids and actual unit-load processing times for the cases are given, for the fast network, in Tab. 3 and, for the slow network, in Tab. 4.

We investigate the effects of cheating on the makespan. Of the eight cases, case (1) results in the optimal makespan; Every other case has a longer makespan. In cases (3), (4), and (5), $P_1$ is under-allocated. In this scenario, the mechanism compensates by allocating more load to the other processors. Processor $P_1$ finishes earlier than the other processors and thus, the makespan is determined by any processor except for $P_1$. In cases (2), (6), (7), and (8), $P_1$ is over-allocated. Consequently, its execution determines the makespan as it finishes later than the others.

Figure 7 and Fig. 8 show the makespan for the eight cases for the fast and slow networks.
balanced binary tree network (Fig. 4), respectively. The following numbers are for the fast network; a comparison of the fast network to the slow network will be given later on. As designed, the minimum makespan is achieved in case (1) when \( t_1 = w_1 = \tilde{w}_1 \) with a makespan of 44.3. The over-allocated \( P_1 \) in case (2) results in a 98.9% increase over the optimal makespan. For cases (3), (4), and (5), the makespan increases by 13.0%. These cases have the same makespan as the under-allocated \( P_1 \) declares the same \( w_1 \) and executes faster than it declares. In the remaining cases, \( P_1 \) is over-allocated. In case (6), the makespan is 63.6%, case (7) is 97.0%, and case (8) is 293% longer than the optimal. The optimal makespan for the slow network is 48.5. Case (2) is 49.5%, case (3) and case (4) are 3.2%, case (5) is 5.0%, case (6) is 37.3%, case (7) is 81.7%, and case (8) is 172% longer than the optimal makespan. It can be noted that for the same magnitude of change in a parameter, the effects of the change is smaller in the slow network than in the fast network.
Figure 11: Completion time when \( P_1 \) cheats in the left unbalanced binary tree network.

Figure 12: Completion time when \( P_1 \) cheats in the right unbalanced binary tree network.

Figure 13: The fast \( P_1 \)'s payment, \( Q_1 \), and utility, \( U_1 \), for the left unbalanced tree network.

Figure 14: The fast \( P_1 \)'s payment, \( Q_1 \), and utility, \( U_1 \), for the right unbalanced tree network.

The utility and payment to \( P_1 \) is shown in Fig. 9 (fast network) and Fig. 10 (slow network). Processor \( P_1 \) maximizes its utility in case (1) in which \( t_1 = w_1 = \tilde{w}_1 \). For the fast \( P_1 \), the utility of case (1) is at least 12.0% greater than the other seven cases. The utility is greatly diminished when \( P_1 \) is over allocated (cases (2), (6), (7), and (8)). These cases exhibit a minimum 58.5% decrease in utility. In particular, case (8) exhibits a dramatic 268% decrease. The network with a slow \( P_1 \) exhibits similar behavior, having a minimum reduction of 3.6% and a maximum reduction of 189%. The magnitude of the percentage decrease is smaller in the slow networks as \( P_1 \) has less power.

We now examine the case when the network is an unbalanced binary tree. Figure 11 and Fig. 12 show the makespan for the networks shown in Fig. 5 and Fig. 6, respectively, using the parameters for the slow network. As we expected, the optimal makespan is obtained in case (1). Furthermore, note that both networks have an optimal
makespan of 48.7. This is a consequence of network symmetry. Comparing cheating in the two networks, the right unbalanced binary tree has a lengthier makespan. The makespan, though, increases by no more than one percent. The difference in makespan is explained by the number of descendants. Having more descendants attenuates the consequences of cheating.

The payment $Q_1$ and utility $U_1$ for the left unbalanced tree and the right unbalanced tree networks using the slow network parameters are given in Fig. 13 and Fig. 14, respectively. As designed, $P_1$ earns the greatest utility when $t_1 = w_1 = \tilde{w}_1$. Processor $P_1$ has more power in the left unbalanced tree than either the balanced or the right unbalanced networks. It earns at least 32.9% more than the $P_1$ in the balanced network and 52.5% more than the $P_1$ in the right unbalanced network. Again this is due to the power of $P_1$. In the cases where it is over-allocated, a greater number of descendants dilutes the effects of cheating and increases the level of utility earned by $P_1$.

We now measure the effect of cheating on the cheater’s utility. We randomly generate 1000 networks with 65 ($m = 64$) processors. In every network, $t_0 = 1000$ and $z_i = 1, i = 1, 2, \ldots, m$. For the non-root processors $P_i \neq P_0$, $t_i$ is drawn from a uniform distribution on the interval $[100, 1000]$. For each network we uniformly and randomly select a processor $P_i$ to cheat. We draw $w_i$ from a uniform distribution on the interval $[100, 1000]$ and $\tilde{w}_i$ from the uniform distribution on the interval $[t_i, 1000]$. For each processor $P_j \neq P_i$, we set the reported and actual unit-load processing times equal to the true value. We measure the percent loss for the cheater in each of the networks. This metric is defined as $-100 \frac{U_i^{\text{cheating}} - U_i^{\text{truthful}}}{U_i^{\text{truthful}}}$, where $U_i^{\text{cheating}}$ is the utility $P_i$ earned from cheating and $U_i^{\text{truthful}}$ is the profit when $P_i$ is truthful. Because the mechanism penalizes the cheaters, this metric will always be greater than or equal to zero. Furthermore, losses can be greater than 100 percent as a cheater may need to pay, resulting in a non-positive utility. For the networks considered here, the minimum and maximum decrease in utility is 0.01% and 7044% respectively. The minimum is the consequence of similar reported and actual unit-load processing times that are close to the true unit-load processing time; the maximum is a result of a large difference between a small reported unit-load processing time and a high actual unit-load processing time. When we aggregate the results we obtained a mean of 205% and a
median of 45.7% for the utility losses. These large losses will prevent cheating by bounded rational agents (processors).

All these experiments show that our mechanism induces truthful reporting and execution. The processors are not able to increase their utility by cheating. They obtain maximum utility only if they report their true parameters and execute the load at their true processing capacity.

7. Conclusion

We proposed DLS-T a distributed mechanism for scheduling divisible loads in tree networks. In a tree network, the load originates at the root and it is dispersed through intermediate processors until the load is distributed to all. The mechanism computes the optimal load allocations as an ex post Nash equilibrium. A consequence of this is that rational processors will choose to follow the suggested strategy (i.e., the algorithm) and report their true parameters as these will maximize their utility. Any deviation by a single processor reduces its utility. For the strongly indifferent selfish-and-annoying agents, the proposed solution bonus is effective in incentivizing these agents to follow the suggested strategy. We further prove that at the equilibrium, the agents obtain non-negative utility. Finally, we demonstrate the properties of DLS-T by simulation.

We are planning to continue our investigation into the augmentation of DLS theory with incentives. By examining various influences such as network architectures and the rational behavior of links, we hope to achieve a cohesive theory combining DLS with incentives. Additionally, we plan to study the incentives that drive the network organization and to design mechanisms for self-organizing DLS.

Acknowledgments

This paper is a revised and extended version of [36] presented at the 20th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2006). The authors wish to express their thanks to the editor and the anonymous referees for their helpful and constructive suggestions, which considerably improved the quality of the paper. This research was supported, in part, by NSF grants CNS-1116787 and DGE-0654014.
References


37


