Probabilistic Byzantine Tolerance Scheduling in Hybrid Cloud Environments

Luciana Arantes, Roy Friedman, Olivier Marin, Pierre Sens

Probabilistic Byzantine Tolerance Scheduling in Hybrid Cloud Environments

Luciana Arantes[*1], Roy Friedman[†2], Olivier Marin[‡3], and Pierre Sens[§1]

[1]Sorbonne Universités, UPMC, CNRS, Inria, LIP6
[2]Computer Science, Technion
[3]Computer Science, NYU Shangai

November 21, 2016

[*]Luciana.Arantes@lip6.fr
[†]roy@cs.technion.ac.il
[‡]ogm2@nyu.edu
[§]Pierre.Sens@lip6.fr

Abstract

This work explores scheduling challenges in providing probabilistic Byzantine fault tolerance in a hybrid cloud environment, consisting of nodes with varying reliability levels, compute power, and monetary cost. In this context, the probabilistic Byzantine fault tolerance guarantee refers to the confidence level that the result of a given computation is correct despite potential Byzantine failures. We formally define a family of such scheduling problems distinguished by whether they insist on meeting a given latency limit and trying to optimize the monetary budget or vice versa. For the case where the latency bound is a restriction and the budget should be optimized, we present several heuristic protocols and compare between them using extensive simulations.

1 introduction

High performance distributed computing (HPDC) is typically obtained by breaking large computational problems offering trivial parallelism into multiple independent compute tasks and scheduling each task to be executed in a distributed environment. This enables solving heavy computational tasks using commodity hardware and operating systems.

With cloud technology, it is common to submit computations to virtual machines hosted in the cloud. The benefit of clouds includes their increased dependability and availability. However, cloud usage for heavy computations involves substantial financial costs.

This motivates exploring hybrid computing architectures that combine desktop Grids with cloud hosted computing. In such a system, a large fraction of the computation is performed by donated machines, which significantly reduces the cost to the owner of the computation. Yet, when donated machines fail to ensure timely reliable completion, parts of the computation can be transferred to the cloud where they are ensured to obtain enough resources to complete.

Unfortunately, donated computers suffer from a non-negligible probability that some of them will not always return correct answers, i.e., act in a Byzantine manner. Such behavior might result, e.g., from malice on behalf of the owner of the machine, from intrusions to the machine, or from faults and bugs in hardware and software.

On the other hand, cloud servers and their VMs are likely to be more dependable than regular home machines. This is because cloud providers have a monetary incentive to protect their infrastructure from intrusions and to maintain their hardware and operating systems up to date. However, these cannot completely rule out intrusions and other forms of Byzantine hardware.

Finally, with recent trusted computing hardware, cloud providers can maintain a smaller set of fully trusted machines. Given the higher cost of trusted computing hardware, it is sensible to assume that these nodes will be scarce and their usage will be considerably more expensive than using standard cloud nodes.

Our contributions First, we define a probabilistic hybrid computing model for HPDC composed of both home donated machines and cloud nodes. In this model, each computational task has a minimal required reliability level, a latency bound, and a budget. Similarly, each node has a known computing speed, monetary cost, and reliability reputation. We distinguish between home donated nodes which are very cheap, but are also not very reliable, standard cloud nodes which are much more dependable and powerful, but are more expensive, and cloud fully trusted nodes, which are completely dependable, but much more costly than the others and are slower than the standard cloud nodes.

Second, we define corresponding scheduling optimization problems that need to ensure that computational tasks meet their requirements. In all of them, the scheduler's goal is to find compute nodes that can return a reply whose correctness is above a given reliability threshold. However, they vary in the latency and budget guarantees they provide. One set of problems ensure bounded latency and attempt to minimize the required budget while the others ensure a bounded budget and try to minimize the latency.

Third, we devise several heuristic protocols for these problems. For lack of space, this paper focuses on solving the variant in which the latency bound must be observed while trying to minimize the required budget.

Last, we evaluate the performance of our protocols by simulation. Results show that in all tested workloads, an adaptive protocol, which schedules tasks on different groups of nodes, is efficient and successfully allocates all tasks within the latency constraints.

The rest of this paper is organized as follows: We survey related work in Section 2. Model assumptions are defined in Section 3. The problem statements as variations of optimization problems are presented in Section 4. Section 5 describes our scheduling protocols for bounded latency. Section 6 presents performance evaluation results of these protocols through extensive simulations. Finally, Section 7 concludes the paper.

2 Related Work

There is vast literature on *Byzantine fault tolerance* (BFT), mainly w.r.t. *consensus* and *state replication*, e.g., [8, 12, 13, 15, 17] to name a few. In particular, multiple works have studied the notion of probabilistic consensus, where either safety is always ensured and termination becomes probabilistic or vice versa, e.g., [5–7, 10, 16, 21, 24, 25, 30]. Such level of BFT requires a minimum of $3f + 1$ replicas, resulting in significant resource overheads, especially when f can be larger than 1.

Reducing this inherent overhead has been explored in several ways. One example is the introduction of wormholes and other types of trusted hardware components [31]. Another idea is separating ordering from execution as proposed in [32], such that the data itself is replicated only on $2f + 1$ nodes. Yet, this requires a separate ordering service, often implemented by traditional BFT protocols, so the savings depends on data being much larger than control and meta-data sizes.

Several works have explored how to harden high performance distributed computing environments, such as [2, 3], against Byzantine failures [1, 9, 11, 28, 29]. Our model differs from theirs in the following ways: At the node level, we incorporate in our model the reliability level of each node, the computational power of each node, and the monetary cost of using each node. In particular, we assume a hybrid execution model composed of nodes from several levels of reliability, compute power, and cost. Further, at the computational task level, we combine a minimal reliability level, latency constraints, and budget constraints. Hence, the scheduling task in our work is much more involved.

A pull-based scheduler for hybrid distributed computing infrastructure (Desktop, Grid, and Cloud nodes) that relies on multi-criteria decision-making method for task assignment is presented in [23]. The multi-criteria method computes for each task a set of criteria according to the characteristics of the node requesting a task. Among the criteria, the authors propose the expected time and cost to complete the task, as well as the the estimated error impact of scheduling the task to the pulling node, taking into account both its reputation and the size of the task. Similarly to us, their aim is to find an optimal scheduling strategy in a hybrid environment that satisfies user constraints expressed in terms of cost, price, and reliability. To this end, they apply a filtering methodology denoted SOFT. Yet, contrarily to our approach, their scheduler is pull-based where idle nodes ask for task assignment, there is no Byzantine behavior, and the scheduler does not apply a per group selection approach as our adaptive protocol does in order to avoid combinatory explosion.

Some hybrid Cloud platforms combine public and private Clouds. Some works [19,20] focus on resource provisioning in the presence of failures. Private cloud nodes are usually considered free for users but prone to failures whereas public cloud nodes are trusted (failure-free) but users must pay to use them. As in our approach, task scheduling decisions depend on reliability requirements and cost constraints. For instance, in [20], the authors consider that failures in private cloud can be correlated in space and time. Hence, jobs that request more than a number of VM threshold or last more than a deadline threshold are redirected to public cloud nodes.

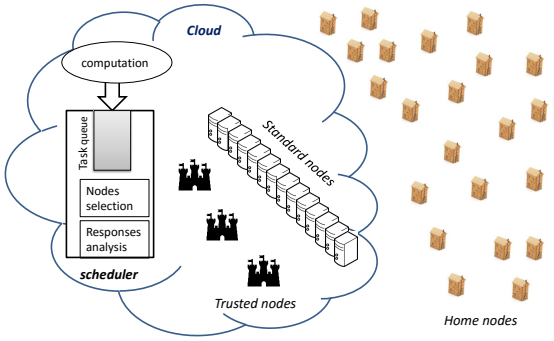

Figure 1: Hybrid cloud architecture

p	A compute node	$B(\Gamma_\tau)$	Budget of schedule Γ_τ
p_i, p_j	Distinguishing between compute nodes	Δ_τ	A latency threshold for τ
q	A probability	Φ_τ	A budget threshold for τ
q_p	Probability that p will return an incorrect answer	$G_\mathcal{T}$	Group of fully trusted nodes
r_p	Reliability of p $(1 - q_p)$	$G_\mathcal{C}$	Group of standard cluster nodes
c_p	Compute speed of node p	$G_\mathcal{H}$	Group of home nodes
d_p	Cost of node p	G_i	A group of nodes of the same type
τ	A compute task	σ	A run
T_τ	Normalized compute time for τ	L_σ	The latency of σ
ρ_τ	A reliability threshold for τ	B_σ	The latency of σ
Γ_τ	A schedule for τ	S_σ	Set of nodes producing replies in σ
$V(\Gamma_\tau)$	A reply values sequence for Γ_τ	t_i	Time
$L(\Gamma_\tau)$	Latency of schedule Γ_τ	v_i	Reply value

Table 1: Main symbols used in this paper

3 System and Threat Models

We consider a hybrid cloud architecture in which *computing tasks* continuously arrive and need to be scheduled on a large pool of available *compute nodes* (physical or VMs), similar to the one depicted in Figure 1. The compute nodes include a combination of *home donated desk-tops* (a.k.a *home nodes*) as well as *cloud hosted VMs* (a.k.a *cloud nodes*). The cloud nodes are also divided into a plurality of *standard cloud nodes* and a smaller set of *fully trusted cloud nodes*. The difference between these sets of nodes is relates to their compute power, expected reliability, and availability.

That is, home nodes are expected to have relatively lower compute power and high *churn rate*, meaning that their availability and reliability are low. In particular, the probability of Byzantine behavior on their part is higher than all other nodes. Usage of home nodes is assumed to be very cheap, but not completely free of charge, as typically cloud providers charge a small amount of money for I/O outside the cloud.

Standard cloud nodes have relatively high compute power and can be allocated on demand. They are much more reliable than home nodes, but they might still occasionally fail including in a Byzantine manner. Utilizing standard cloud nodes involves paying a small fee.

Finally, fully trusted cloud nodes have a very low probability of suffering a *crash* failure, and never suffer from Byzantine failures. The number of fully trusted cloud nodes is bounded and their usage cost is significantly higher than using standard cloud nodes. The existence of such fully trusted nodes is backed by recent advancements in trusted cloud computing hardware [31], or alternatively can be realized by clustering standard cloud nodes using BFT techniques [8, 12, 13].

Further, the scheduler, which accepts the computing tasks and distributes them to various compute nodes, runs on a fully trusted cloud node. That is, the scheduler is assumed to be fault-tolerant, always available, and it always obeys its prescribed protocol.

The communication in the system is performed by sending and receiving messages over a communication network. The network is assumed to be authenticated and reliable, with a bounded communication latency. That is, a message is only received if it was indeed sent by some node, and the receiver always knows who the true sender of a message is.

As mentioned before, the home nodes and standard cloud nodes may occasionally act in a *Byzantine* manner. That is, while executing a compute task, each such node p may return an incorrect answer (or not answer at all) with probability q_p. We refer to the probability $r_p = 1 - q_p$ that p returns a correct answer as the *reliability* or *reputation* of p. Notice that r_p may change overtime. When the system starts, for any home node p_i and arbitrary standard cloud node p_j, $r_{p_i} < r_{p_j}$. Further, even when the reliability of nodes changes over time, the above inequality would remain true for the vast majority of home nodes and cloud nodes. Obviously, for a fully trusted node p, r_p

is always 1.

Whenever a scheduler node receives a compute task, it sends it either to a fully trusted cloud node or to multiple compute nodes. In the former case, the scheduler knows that it will get a correct answer. However, since fully trusted nodes are scarce and expensive, the scheduler often prefers the latter option. In these cases, when the replies arrive, the scheduler compares them. If they all agree, then the scheduler knows that this is the correct answer with a certainty that depends on the reputations of the chosen nodes. Otherwise, if some replies do not return within the deadline, the scheduler knows that these nodes are faulty and sends the same compute task to additional nodes. Yet, if the replies do not match, then the scheduler knows that at least some of the nodes acted in a Byzantine manner and may send the compute task to additional nodes until it has enough probabilistic confidence in one of the replies.

Each compute task τ has a normalized compute time T_τ and that each compute node p has a known computing speed c_p. So when there are no failures, a task τ that is scheduled to be computed on a node p completes its execution on p within time T_τ/c_p. Further, each compute node p charges d_p units of money per second of computing (for home nodes $d_p = 0$). Hence, the cost of computing task τ on node p is $\frac{T_\tau \cdot d_p}{c_p}$.

The number of nodes needed to execute each compute task to obtain a trusted reply as well as the expected compute time and cost are the main topic of this paper.

3.1 A Generic Model of Hybrid Computation

In the most generic case, compute nodes are divided into multiple groups $\{G_i\}$, each characterized by an initial reputation value R_i, its own scheme for managing the reputation by the scheduler F_i, a range of compute power $[C_i^{\min}, \ldots, C_i^{\max}]$, and a range of costs $[D_i^{\min}, \ldots, D_i^{\max}]$. For each group G_i and for each node $p \in G_i$, the initial reputation of p, $r_p = R_i$. The compute power $c_p \in [C_i^{\min}, \ldots, C_i^{\max}]$ and the computing cost $d_p \in [D_i^{\min}, \ldots, D_i^{\max}]$. The reputation management scheme F_i is a function that is invoked by the scheduler each time a computation that p was involved in terminates and sets a new value for r_p based on its old value and all returned results for the compute task.

In the case of home machines, standard cloud nodes, and fully trusted cloud nodes, each of the above forms a group. For the fully trusted nodes $R_i = 1$ and $F_i(*) = 1$. Further, $D_i^{\min} > D_k^{\max}$ for any other k, i.e., they are the most expensive nodes. However, $C_i^{\max} < C_k^{\min}$ when k is standard cloud nodes, as trusted hardware is assumed to be slower than commodity one.

At the other end, in the desktop group, $R_i < R_k$ for any other group k, i.e., these nodes are the least trusted. Also, $D_i^{\min} \leq D_i^{\max} << D_k$ for any other k, i.e., using these nodes is very cheap compared to the others.

Several works explored the reputation management schemes (F_is) and its effectiveness, e.g., [4,18,26,28]. For lack of space, we defer exploring the impact of F_i to future work. In particular, the simulations in Section 6 assume that each node has a fixed reliability level.

4 Problem Statement

We present two variations of an optimization problem. Specifically, the goal of the scheduler is to submit each task to one or more compute nodes such that the reliability level of the result is equal or greater to a given threshold ρ. In one variant of the problem, each application, composed by a set of tasks, gets a maximum budget allocation, and the scheduler needs to minimize the expected latency until obtaining a correct answer. In the other variant, the application has a maximal latency and the scheduler needs to minimize the cost of the computation. Yet, in order to rigorously state the problem definitions, we must first develop adequate terminology. In the definitions below, for simplicity of presentation, we ignore the transmission times of messages.

4.1 Framework and Terminology

4.1.1 Schedules

Given a compute task τ, a *schedule* Γ_τ is a sequence of tuples $\{< t_i, p_i >\}$ representing a time t_i and a compute node p_i such that for each two tuples $< t_i, p_i >$ and $< t_j, p_j >$ in Γ_τ, if $p_i \neq p_j$ and $< t_i, p_i >$ appears before $< t_j, p_j >$ in Γ_τ then $t_i \leq t_j$. Intuitively, the schedule indicates the starting time of the task on each compute node.

Since we assumed that each task τ has a normalized compute time T_τ and each node p has a compute power c_p, each tuple in Γ_τ is implicitly associated with a time $t'_i = t_i + \frac{T_\tau}{c_p}$ such that if p is correct, the scheduler is guaranteed to obtain a reply from p by t'_i. In fact, if no reply is received by t'_i, then p is assumed to be faulty.

4.1.2 Reply Sequences

Suppose τ is sent to various compute nodes according to Γ_τ. The replied values can be represented by a sequence V_{Γ_τ} of tuples $< t'_i, v_i >$ where v_i is the corresponding reply value arriving at time t'_i. If a reply does not arrive in time, the matching v_i is set to $\perp$. In case p_i is correct, v_i is the correct reply. A reply value sequence V_{Γ_τ} is called *complete* if for each tuple in Γ_τ there is a corresponding tuple in V_{Γ_τ} and is said to be *partial* if it is a prefix of a complete reply sequence.

4.1.3 Runs

Next, we define a boolean *stopping function* $ST(V_{\Gamma_\tau}) = [\texttt{true}, \texttt{false}]$. Intuitively, this function enables the scheduler to stop the schedule prematurely, before contacting all nodes indicated by the schedule, whenever the replies obtained so far meet the condition indicated by the stopping function. To that end, we define a *stoppable schedule* to be the combination of a schedule and a corresponding stopping function. Hereafter, we only deal with stoppable schedules. Hence, whenever we write schedule, we in fact mean stoppable schedule.

A (partial) reply sequence V_{Γ_τ} is called *minimal* if $ST(V_{\Gamma_\tau}) = \texttt{true}$ and for each prefix V' of V_{Γ_τ}, $ST(V') = \texttt{false}$. Each combination of a schedule and a minimal reply sequence defines a *run* σ.

4.1.4 Latencies and Budgets

Let $< t_1, * >$ be the first tuple in a schedule of a run σ and let $< t_k, * >$ be the last tuple in the matching reply sequence of σ. We define the *latency* of σ (denoted L_σ) to be $t_k - t_1$. Similarly, we define the *budget spent during* σ (denoted B_σ) as the total cost of all compute nodes whose replies appear in the corresponding reply sequence V_{Γ_τ}. That is, let S_σ be this set of compute nodes. We then have $B_\sigma = \sum_{p \in S_\sigma} \frac{T_\tau d_p}{c_p}$.

4.2 Two Families of Problems

We define two families of dual optimizations problems. In the first, latency must be kept below a given bound while the budget should be minimized. In the second, the budget must be kept below a given bound while latency should be minimized.

4.2.1 Bounded Latency

Given a task τ, a latency threshold Δ_τ and reliability threshold ρ_τ for task τ, the scheduler's goal is to find a schedule Γ_τ minimizing the corresponding budget $B(\Gamma_\tau)$ for computing τ while obtaining an answer whose reliability is above ρ_τ and the latency $L(\Gamma_\tau)$ is at most Δ_τ time (assuming feasible).

4.2.2 Bounded Budget

Given a task τ, a budget threshold Φ_τ and reliability threshold ρ_τ for task τ, the scheduler's goal is to find a schedule Γ_τ minimizing the corresponding latency $L(\Gamma_\tau)$ for computing τ while obtaining an answer whose reliability is above ρ_τ and the budget $B(\Gamma_\tau)$ is at most Φ_τ (assuming feasible).

As mentioned before, due to lack of space, in this paper we focus on solving the bounded latency problem.

5 Greedy Bounded Latency

In this section, we present several variants of scheduling protocols addressing the bounded latency problem. The first set of protocols always attempt to find a schedule from the same group of nodes (home, cloud, or trusted) in an iterative manner. Conversely, the other protocol employs an adaptive mechanism in which it considers all groups and chooses the cheapest option that still ensures timely termination.

5.1 Single Group Protocols

We identify three protocols in the single group category, nicknamed *scrooge*, *moderate*, and *cautious*. The only difference between these protocols is in the group of processes from which each of these protocols looks for its candidate nodes; scrooge only utilizes home nodes, moderate only accesses standard cloud nodes, while cautious only fully trusted nodes.

In all three protocols, the scheduler invokes the following iterative loop, as outlined in Algorithms 1, 2, 3, and 4: First, it looks for the cheapest set of available nodes from the corresponding group (home, standard cloud, or trusted) such that the probability that all of them are Byzantine is below the required reliability threshold and the maximal latency of any of them for the given compute task is below its latency requirement. The task is then sent to all chosen nodes to be computed and the scheduler waits until it receives either a reply from all of them or a timeout equal to the longest expected latency has passed. If all replies have arrived and all have the same value, then the scheduler returns this value.

Otherwise, suppose some value v has appeared in the maximal number of replies (breaking symmetry arbitrarily). The scheduler looks for another set of processes such that if they all return v, then the probability that v is the correct value is above the required threshold. Here again, the cheapest possible set that meets the remaining latency deadline is chosen. This process repeats until either the probability that some returned value is correct meets the reliability threshold, or it is not possible to find additional nodes that can compute the task within the required deadline.

Algorithm 1 Probabilistic Functions

```
1: // Returns TRUE if the probability that all nodes in S are Byzantine is ≤ q
2: function AllByz(S, q)
3:     return ∏_{p_i∈S}(1 − r_i) ≤ q
4:
5: // Returns the probability that at least one of the nodes in values whose value is v is correct and all nodes proposing other values are Byzantine
6: // Also, return the needed probability from additional set of supporters of v to ensure that v is correct if not already obtained
7: function ProbCorrectValue(v, values, q)
8:     if v == ⊥ then
9:         return (0,0)
10:    S_1 = {p_i|(v_i, p_i) ∈ values ⋀ v_i == v}
11:    AtLeastOneCorrect = 1 − ∏_{p_i∈S_1}(1 − r_i)
12:    S_2 = {p_i|(v_i, p_i) ∈ values ⋀ v_i ≠ v}
13:    AllByzantine = ∏_{p_i∈S_2}(1 − r_i)
14:    ProbCorrectV = AtLeastOneCorrect · AllByzantine
15:    if ProbCorrectV < q then
16:        ExtraNeeded = (AllByzantine − q) / (AllByzantine − ProbCorrectV)        ▷ See explanation in text
17:        return (ProbCorrectV, ExtraNeeded)
18:    else
19:        return (ProbCorrectV, 0)
```

To understand the calculation for *ExtraNeeded* in Algorithm 1, let's denote *ProbCorrectV* by A, *AllByzantine* by C, and the calculation $\prod_{p_i \in S_1}(1-r_i)$ by D. Obviously, we can write $A = (1-D)C$. Similarly, we can write $q = (1-DX)C$ where X is the probability that all nodes in the added set (that is required) are Byzantine. In other words, $1-DX$ is the probability that the combined set of existing nodes that support v and all added nodes will include at least one correct node. Using simple algebra, we get that $X = \frac{C-q}{C-A}$, or as written in Figure 1, $\mathit{ExtraNeeded} = \frac{\mathit{AllByzantine}-q}{\mathit{AllByzantine}-\mathit{ProbCorrectV}}$.

5.2 The Adaptive Protocol

As before, in the adaptive algorithm too we allow multiple sequential invocations of the task until some returned value obtains enough reliability to be considered the correct reply value. However, here we allow to switch between the different groups of nodes (home, cloud, trusted). That is, home nodes are the cheapest, but may not be able to provide a reliable answer within the deadline due to their low reliability. Trusted nodes always return a correct answer,

Algorithm 2 Helper Functions

```
1: // Add the corresponding budget and latency to a schedule
2: function FleshOut(S, T)
3:     budget = Σ_{p_i∈S} Td_i/c_i
4:     latency = max_{p_i∈S} T/c_i
5:     return (S, budget, latency)
6:
7: // Find a set that can compute T fast enough whose members
   satisfy the required reliability
8: function FindSet(G, T, Δ, q)
9:     cands = {p_i ∈ G | T/c_i ≤ Δ}  ▷ find all nodes that are fast
   enough
10:    if cands == ∅ then
11:        raise no_schedule_found
12:    else
13:        sets = {S_i ⊆ cands | AllByz(S_i, q)} ▷ All possible sets
14:        finalists = {S_i ∈ sets | Σ_{p_i∈S_i} T_τ d_i/c_i is minimal in sets}
   ▷ Filter for the cheapest
15:        if finalists == ∅ then
16:            raise no_schedule_found
17:        else
18:            return FleshOut(first(finalists, T))  ▷ Return
   one of the cheapest
19:
20: // Send T to all members of set and wait for replies
21: function ScheduleStep(set, T, latency)
22:    foreach p_i ∈ set send(p_i, T)
23:    wait for replies from each p_i ∈ set or timeout after latency
   time
24:    replies = set of tuples (v_i, p_i) of replied values and corre-
   sponding nodes who returned these values
25:    for nodes p_i that failed to return any value by the latency
   timeout, the value is v_i = ⊥
26:    return replies
```

Algorithm 3 Parameterized Single Group

```
1: // Schedule T whose deadline is Δ on nodes from G with
   reliability ≥ ρ
2: function Schedule(G, T, Δ, ρ)
3:     values = ∅
4:     reqrel = 1 − ρ
5:     loop
6:         try (set, budget, latency) = FindSet(G, T, Δ, reqrel)
7:         catch no_schedule_found raise no_schedule_found
8:         values = values ⋃ ScheduleStep(set, T, latency)
9:         if |values| == 1 ⋀ this value ≠ ⊥ then
10:                ▷ If all nodes replied the same value, return it
11:            return first(first(values))
12:        else
13:                ▷ If the probability that the majority value v is
   correct is high enough, return it. Otherwise, try collecting additional
   answers from the cheapest set of additional nodes from G such that they
   can return the reply in the remaining time before the deadline and if
   they all return v as well, it will be certain that v is correct
14:            v = the most frequent value in values
15:            Δ = Δ − latency
16:            (rel, reqrel) = ProbCorrectValue(v, values, ρ)
17:            if rel ≥ ρ then
18:                return v
```

Algorithm 4 The Specific Single Group Instances

```
1: function Scrooge(T_τ, Δ_τ, ρ)
2:     return Schedule(G_H, T_τ, Δ_τ, ρ)
3:
4: function Moderate(T_τ, Δ_τ, ρ)
5:     return Schedule(G_C, T_τ, Δ_τ, ρ)
6:
7: function Cautious(T_τ, Δ_τ, ρ)
8:     return Schedule(G_T, T_τ, Δ_τ, ρ)
```

but are expensive and scarce, and hence are likely to run out if we insist on only using them. Cloud nodes are more reliable than home nodes, but might still fail and are considerably more expensive than the home nodes.

Hence, we start with the cheapest set among all groups that is likely to produce a correct result. But, this time, we reserve enough latency so that in the worst case, we can resort to a trusted node, or to at least a collection of cloud nodes, in order to ensure reliable termination.

Obviously, trying all possible combinations of nodes to determine the optimal one is too expensive. Hence, in each iterative step we always select sets of nodes from the same G_i group. Moreover, we identify a few subsets in each group and try combinations of these subsets such that the total latency is below the threshold while the ensured reliability is above the threshold. We then choose the set whose cost is minimal among them.

Specifically, in each iteration we first identify the cheapest and fastest available trusted nodes that can compute T_τ within the remaining deadline Δ'. Obviously, in the first iteration, the above is done w.r.t. the entire deadline, i.e., $\Delta' = \Delta$. Denote the faster of these nodes $p^F_{\mathcal{T}}$ and the cheaper $p^E_{\mathcal{T}}$ (so F stands for fast and E for economic and $\mathcal{T}$ for trusted). Similarly, denote the latency and budget that would be spent when executing on $p^F_{\mathcal{T}}$ by $L^F_{\mathcal{T}}$ and $B^F_{\mathcal{T}}$ respectively. In symmetry, denote the latency and budget that would be spent when executing on $p^E_{\mathcal{T}}$ by $L^E_{\mathcal{T}}$ and $B^E_{\mathcal{T}}$ respectively.

Considering these two latencies, we now identify sets of standard Cloud nodes that can potentially compute T_τ with the required reliability (if all return the same value) within $\Delta'_F = \Delta' - L^F_{\mathcal{T}}$ and

$\Delta'_E = \Delta' - L^E_{\mathcal{T}}$, respectively. For each of these corrected latency bounds, we search for the cheapest and fastest sets of cloud nodes. Denote these sets of cloud nodes $S^{FF}_{\mathcal{C}}$, $S^{FE}_{\mathcal{C}}$, $S^{EF}_{\mathcal{C}}$, and $S^{EE}_{\mathcal{C}}$. Similarly, we denote the corresponding latency and budget to be spent by each of these sets by $L^{FF}_{\mathcal{C}}$, $B^{FF}_{\mathcal{C}}$, $L^{FE}_{\mathcal{C}}$, $B^{FE}_{\mathcal{C}}$, $L^{EF}_{\mathcal{C}}$, $B^{EF}_{\mathcal{C}}$, $L^{EE}_{\mathcal{C}}$ and $B^{EE}_{\mathcal{C}}$.

Finally, for each of the above latencies, we identify sets of home nodes that can potentially compute T_τ with the required reliability level (if all return the same value) within $\Delta'_{FF} = \Delta'_F - L^{FF}_{\mathcal{C}}$, $\Delta'_{FE} = \Delta'_F - L^{FE}_{\mathcal{C}}$, $\Delta'_{EF} = \Delta'_E - L^{EF}_{\mathcal{C}}$, and $\Delta'_{EE} = \Delta'_E - L^{EE}_{\mathcal{C}}$, respectively. Here, it is enough to identify the cheapest such sets of home nodes, denoted $S^{FF}_{\mathcal{H}}$, $S^{FE}_{\mathcal{H}}$, $S^{EF}_{\mathcal{H}}$, and $S^{EE}_{\mathcal{H}}$. Their corresponding latency and budget are denoted $L^{FF}_{\mathcal{H}}$, $B^{FF}_{\mathcal{H}}$, $L^{FE}_{\mathcal{H}}$, $B^{FE}_{\mathcal{H}}$, $L^{EF}_{\mathcal{H}}$, $B^{EF}_{\mathcal{H}}$, $L^{EE}_{\mathcal{H}}$ and $B^{EE}_{\mathcal{H}}$.

Considering $p^F_{\mathcal{T}} = S^F_{\mathcal{T}}$ and $p^E_{\mathcal{T}} = S^E_{\mathcal{T}}$, once we have all the 10 sets, the scheduler picks the one whose budget is smallest, sends the computation to that set and waits either for all replies or a timeout.The scheduler then checks if some value has enough support to be deemed correct with the required reliability threshold. If yes, then this value is returned. Otherwise, the scheduler continues to the next iteration. The pseudo-code for this protocol appears in Algorithms 5 and 6.

Algorithm 5 Helper Function for Adaptive Algorithm

```
1: // Returns the fastest and cheapest sets of nodes from group G than can compute T with reliability ≥ ρ and latency ≤ Δ
2: function FIND2SETS(G, T, Δ, ρ)
3:     cands = {p_i ∈ G | T/c_i ≤ Δ}  ▷ Find qualifying nodes in G
4:     if cands == ∅ then
5:         return {(∅, 0, 0), (∅, 0, 0)}
6:     else
7:                 ▷ Identify the cheapest set, break ties arbitrarily
8:         sets = {S_i ⊆ cands | ALL-BYZ(S_i, ρ)} ⋀ S_i is minimal such set}
9:         cheapests = {S_i ∈ sets | Σ_{p_i∈S_i} Td_i/c_i is minimal in sets}
10:        if |cheapests| > 1 then
11:            cheapest = S_i | min_{p_i∈S_i} c_i is maximal in cheapests   ▷ break symmetry arbitrarily
12:        else
13:            cheapest = FIRST(cheapests)
14:                ▷ Identify the fastest set, break ties arbitrarily
15:        fastests = {S_i ∈ sets | min_{p_i∈S_i} c_i is maximal in sets}
16:        if |fastests| > 1 then
17:            fastest = S_i | Σ_{p_i∈S_i} Td_i/c_i is minimal in fastests   ▷ break symmetry arbitrarily
18:        else
19:            fastest = FIRST(fastests)
20:            ▷ Return found sets with their budget and latency
21:        if cheapest == fastest then
22:            return {FLESHOUT(cheapest, T),(∅, 0, 0)}
23:        else
24:            return {FLESHOUT(cheapest, T),FLESHOUT(fastest, T)}
```

6 Evaluation

We evaluate our algorithms and show the advantage of the adaptive approach. Our experiments consist of simulating the various protocols described in Section 5 along the metrics defined in Section 6.1 below. The parameters for the simulations are detailed in Section 6.2.

6.1 Metrics

We define the following metrics for comparing the performance of the various protocols: (i) *Budget (financial cost):* the total budget actually spent by the protocol; (ii) *Completion time (response time):* The overall completion time of the application; (iii) *Computation time:* The difference between the time a task is submitted and the time it responds (both the average and the distribution); (iv) *Task deadlines satisfaction (fail ratio):* Percentage of task deadlines that were satisfied w.r.t. latency (deadline) and reliability constraints. A failure of a task is due either to wrong answers from Byzantine nodes which delay the response time or an overload of compute nodes; (v) *Distribution of scheduled nodes:* The total number of home, standard cloud, and fully trusted nodes used by each of the protocols; (vi) *Scheduler compute time:* The amount of CPU time used by the protocol to schedule all the tasks. This metric indicates the computational complexity of the protocol, which is important for scalability.

6.2 Simulation setup

Our simulations are conducted with Matlab[1]. Since the proposed algorithms need to compute all subsets

[1] http://mathworks.com/

Algorithm 6 Adaptive Algorithm

Main code:
$values = \emptyset$
$reqrel = 1 - \rho$
loop
  $(S^E_{\mathcal{T}}, B^E_{\mathcal{T}}, L^E_{\mathcal{T}}, S^F_{\mathcal{T}}, B^F_{\mathcal{T}}, L^F_{\mathcal{T}}) =$ FIND2SETS$(G_{\mathcal{T}}, T_\tau, \Delta, reqrel)$
  $(S^{EE}_{\mathcal{C}}, B^{EE}_{\mathcal{C}}, L^{EE}_{\mathcal{C}}, S^{FE}_{\mathcal{C}}, B^{FE}_{\mathcal{C}}, L^{FE}_{\mathcal{C}}) =$ FIND2SETS$(G_{\mathcal{C}}, T_\tau, \Delta - L^E_{\mathcal{T}}, reqrel)$
  $(S^{EF}_{\mathcal{C}}, B^{EF}_{\mathcal{C}}, L^{EF}_{\mathcal{C}}, S^{FF}_{\mathcal{C}}, B^{FF}_{\mathcal{C}}, L^{FF}_{\mathcal{C}}) =$ FIND2SETS$(G_{\mathcal{C}}, T_\tau, \Delta - L^F_{\mathcal{T}}, reqrel)$
  $(S^{EE}_{\mathcal{H}}, B^{EE}_{\mathcal{H}}, L^{EE}_{\mathcal{H}}, -, -, -) =$ FIND2SETS$(G_{\mathcal{H}}, T_\tau, \Delta - L^{EE}_{\mathcal{C}}, reqrel)$
  $(S^{EF}_{\mathcal{H}}, B^{EF}_{\mathcal{H}}, L^{EF}_{\mathcal{H}}, -, -, -) =$ FIND2SETS$(G_{\mathcal{H}}, T_\tau, \Delta - L^{EF}_{\mathcal{C}}, reqrel)$
  $(S^{FE}_{\mathcal{H}}, B^{FE}_{\mathcal{H}}, L^{FE}_{\mathcal{H}}, -, -, -) =$ FIND2SETS$(G_{\mathcal{H}}, T_\tau, \Delta - L^{FE}_{\mathcal{C}}, reqrel)$
  $(S^{FF}_{\mathcal{H}}, B^{FF}_{\mathcal{H}}, L^{FF}_{\mathcal{H}}, -, -, -) =$ FIND2SETS$(G_{\mathcal{H}}, T_\tau, \Delta - L^{FF}_{\mathcal{C}}, reqrel)$
  $set = S \in \{S^E_{\mathcal{T}}, S^F_{\mathcal{T}}, S^{EE}_{\mathcal{C}}, S^{FE}_{\mathcal{C}}, S^{EF}_{\mathcal{C}}, S^{FF}_{\mathcal{C}}, S^{EE}_{\mathcal{H}}, S^{EF}_{\mathcal{H}}, S^{FE}_{\mathcal{H}}, S^{FF}_{\mathcal{H}}\}$ s.t. $S \neq \emptyset \bigwedge$ the corresponding budget is minimal
                                ▷ L_S denotes the latency of S
  if $set == \emptyset$ **then**
    raise `no_schedule_found`
  $values = values \bigcup$ SCHEDULESTEP(set, T_τ, L_S)
  if $|values| == 1 \bigwedge$ this value $\neq \perp$ **then**
                ▷ If all nodes replied the same value, return it
    return `first(first(`$values$`))`
  else
                ▷ If the probability that the majority value v is correct is high enough, return it. Otherwise, try collecting additional answers from the cheapest set of additional nodes from G such that they can return the reply in the remaining time before the deadline and if they all return v as well, it will be certain that v is correct
    $v =$ the most frequent value in $values$
    $\Delta = \Delta - L_S$
    $(rel, reqrel) =$ PROBCORRECTVALUE$(v, values, \rho)$
    if $rel \geq \rho$ **then**
      return v

of nodes that satisfy a criteria of cost and/or time which may lead to a combinatory explosion, an exhaustive search of all subsets is too costly. For instance, assigning 1,000 tasks with normalized duration of 180 seconds to 300 nodes by the Moderate algorithm takes 2887 seconds on a 2 cores Intel core I7 at 1.8GHz. Inspired by the power of two random choices [22], we circumvent this combinatorial explosion by random sampling of subsets and then choosing among them the one that satisfies the criteria. For instance, the 10 sampling version of the Moderate algorithm generates results which are 93% close to the original algorithm's in less than 0.5 second. Consequently, the performance results shown here correspond to the modified versions of our algorithms that use a 10 random sampling.

Nodes setting We consider 3 sets of nodes (*Trusted*, *Cloud*, and *Home*). Each of them is associated to some computing power and financial cost per hour.

Trusted nodes. These secure cloud nodes do not need replicated execution. Alas, they are more expensive and slower than Cloud nodes. Each Trusted node has the processing power of a medium Home node with 8 ECUs (EC2 Computing Units). One ECU provides equivalent CPU capacity of 1.0-1.2 GHz 2007 Opteron. A Trusted node cost corresponds to a high power node in Amazon (c4.8xlarge), which is 1.763 dollars per hour according to EC2 pricing (https://aws.amazon.com/ec2/).

Cloud nodes. These cloud nodes are less secure but more powerful than Trusted nodes. In our simulations, Cloud nodes consist of three types of EC2 Amazon nodes from medium to high computing performance (c4.large, c4.xlarge, and c4.2xlarge instances) corresponding to Intel Xeon E5-2666v3 with 8, 16, and 31 ECUs respectively. The EC2 computing cost per hour for these node types is 0.11, 0.22, and 0.441 dollars respectively.

Home nodes. These home nodes are cheaper and less trusted than the other two kinds of nodes. Their compute power varies from 2 to 16 equivalent ECUs (2, 4, 8, and 16). To estimate their cost, we consider their electric consumption. We assume that each node consumes 100 Watt per hour and that the cost of energy is between 10 and 40 cents per KW/h (the average cost of energy in north America and Europe). Their resulting costs are 0.01, 0.015, 0.02, and 0.04 dollar per hour.

For our simulation experiments, we consider an environment with 5,000 Home nodes, 500 Cloud nodes, and 50 Trusted nodes. In particular, there are 10 times more Home nodes than Cloud nodes and 10 times more Cloud nodes than Trusted nodes. Further, the percentage of Byzantine nodes in the Home nodes group is higher than in the Cloud nodes group while correct nodes in the latter have higher reputation than in the former. Table 2 summarizes our nodes setting for the three groups of nodes. In the

table, the reputation value of correct is denoted r and Byzantine nodes rb, while the percentage of Byzantine nodes is denoted b.

	Trusted nodes	Cloud nodes	Home nodes
Number	50	500	5000
ECU	8	8/16/31	2/4/8/16
Cost ($/h)	1.763	0.11/0.22/0.441	0.01/0.015/0.02/0.04
b	0	5%	15%
r	N/A	0.99	0.95
rb	N/A	0.2	0.1

Table 2: Hybrid cloud configuration

Workload	# tasks	mean	stddev	min	max
low	5,000	179.9923	4.1528	162.307	194.5323
high	20,000	180.0023	4.1274	162.307	196.209

Table 3: Workload configurations

Algorithm	Time (sec)	Cost ($)	Sched. time (sec.)	fail ratio
Cautious	476.98	11.588	0.041	79 %
Moderate	477.32	11.979	0.555	0.38 %
Scrooge	454.79	3.189	2.153	0 %
Adaptive	443.77	2.884	5.683	0%

Table 4: Time and cost for protocols with low load

Application setting We consider a bag of tasks (BoT) application representative of private Home Grid deployments [14]. The reliability threshold is $\rho_\tau = 0.999$.

We generated two scenarios with different loads with regard to the number of concurrently submitted tasks: low load (5,000 tasks) and high load (20,000 taks), as summarized in Table 3. The duration of tasks follow a normal distribution with a mean duration around 180 seconds on one ECU. Both the mean duration of tasks and the standard deviation are set to be the same as the ones of BoT application deployed on the Home Grid of the University of Notre Dame [27]. The deadline of every task is 2.5 times the completion time on one ECU.

6.3 Low workload evaluation

Table 4 summarizes the cost, completion time, fail ratio, and the time spent by the scheduler for the four algorithms. We observe that Scrooge has the lowest cost among the single group algorithms. Scrooge is competitive since its nodes are cheap and abundant so they can easily handle the load within the deadline constraints. Moderate and Scrooge allocate additional nodes when wrong responses are received: on average, a task is replicated 4.225 times in the case of Home nodes and 2.227 times in the case of Cloud nodes. On the other hand, even if Trusted nodes are never replicated, they cannot ensure the execution deadline of most of the tasks. The adaptive protocol also succeeds to execute all the tasks within their deadline. Compared to the cheapest single protocol, i.e., Scrooge, the response time for the application is 2.48% lower. However, even if the cost of the adaptive protocol is 10.58% smaller than Scrooge, we cannot conclude that the adaptive protocol is always the cheapest since the difference of costs is due to the heterogeneity of nodes and the load distribution.

Table 5 shows the distributions of load in terms of the number of tasks executed by nodes of different groups. A Trusted node executes 21 tasks and then fails to satisfy deadlines. The load is unbalanced in both Cloud and Home nodes since the power of nodes is highly heterogeneous. Powerful nodes execute a large number of tasks. In both Scrooge and Adaptive, Home nodes are not used since the number of tasks is relatively low.

Figure 2 gives the computation time of each task. We observe that only 1,050 tasks are completed before their deadline in the Cautious algorithm, the Moderate algorithm fails to complete 19 tasks, and only Scrooge and Adaptive succeed in completing all tasks. In the adaptive approach, we observe that Trusted and Home nodes execute all the tasks, i.e., no task is assigned to nodes of the Cloud group. Interestingly, even in the relatively low load configura-

Algorithm	# nodes	mean	stddev	min	max
Cautious	50	21	0	21	21
Moderate	500	22.186	11.8977	1	41
Scrooge	4408	4.225	3.5449	0	24
Adaptive	3375	4.232	4.3195	0	31

Table 5: Distribution of tasks on nodes with low load

Algorithm	Time (sec)	Cost ($)	Sched. time (sec.)	fail ratio
Cautious	476.98	11.588	0.115	94.75 %
Moderate	485.35	16.594	1.476	66.23 %
Scrooge	492.94	11.064	3.770	14.04 %
Adaptive	471.52	17.816	13.391	0 %

Table 6: Time and cost with high load

Algorithm	# nodes	mean	stddev	min	max
Cautious	50	21	0	21	21
Moderate	500	30.358	11.997	10	65
Scrooge	4999	14.413	8.4422	0	39
Adaptive					
Trust set	45	2.32	1.53	0	7
Cloud set	377	12.38	8.17	0	31
Home set	4949	14.56	10.29	0	40

Table 7: Distribution of tasks on nodes with high load

tion, the application benefits from the adaptive algorithm since it distributes the load among the cheapest nodes but assigns a task to a Trusted node whenever the execution of this task cannot be performed on the Home group.

6.4 High workload evaluation

Table 6 summarizes the results with heavy load. Since there is no sharing of tasks between the three groups for the single group approach, the three algorithms fail to complete the deadline for a large number of tasks. Having a higher number of nodes (20,000), Scrooge is clearly the most efficient with regard to fail deadline ratio. However, costs are not representative since the algorithms do not succeed in executing all the tasks. The average costs in dollars per scheduled tasks are 0.011, 0.0025, 0.00064 in Cautious, Moderate, and Scrooge respectively. The adaptive algorithm succeeds in executing all the tasks within their deadline. Compared to the single group algorithms, the response time for the application is lower than Scrooge and the global cost is higher since all tasks are scheduled. The average cost per task is 0.00089 dollars. The cost per task is 38,4% higher than with Scrooge since the adaptive scheduler needs to use expensive Trusted nodes in this scenario.

Table 7 provides the tasks distribution per node for each group. The load distributions are similar to the low load scenario. The algorithms favor choosing powerful nodes; even if the latter are more costly, they succeed more often to complete tasks within the deadline. Thus, their short response time reduces costs. Notice that the adaptive algorithm uses all three sets of nodes.

Figure 3 shows the computation time of each task. In the single group approach, a large number of tasks cannot meet their deadline and are, thus, not scheduled. As shown in Figure 3(a), after 1,000 tasks, no other task is scheduled by the Cautious protocol. The gaps without load after the number of scheduled tasks reaches 5,500 in Figure 3(b) indicate that many tasks could not be scheduled in the Moderate protocol. Some tasks, denoted "faulty tasks", are scheduled on Cloud and Home nodes but fail to meet their deadline because of the high number of additional nodes needed to obtain enough correct answers. Among the $6,753$ tasks it schedules, Moderate generates 173 faulty tasks (2.56%) while Scrooge generates 332 faulty tasks among its $17,253$ scheduled tasks (1.92%). Figure 3(d) gives the computation time of each task with Adaptive. We observe that Trust nodes start participating in the computation after 8,000 scheduled tasks. After $15,000$ tasks, Trust and Home nodes become overloaded and then tasks are mainly scheduled on the nodes of the Cloud group. Ultimately, $17,308$ tasks are scheduled on Home nodes, $2,692$ on Cloud nodes, 116 on Trusted nodes. Among these tasks, 109 are "hybrid", i.e., replicated both on Home and Trusted nodes. Figure 4 gives the cumulative number of replicas associated to each set.

7 Conclusion

We have explored a hybrid computing model composed of groups of home, standard cloud, and trusted cloud nodes, where each node has a given computing speed, monetary cost, and reliability reputation. We presented four protocols (Scrooge, Moderate, Cautions, and Adaptive) for scheduling tasks in such environments aiming at ensuring BoT applications' requirements in terms of bounded latency and reliabil-

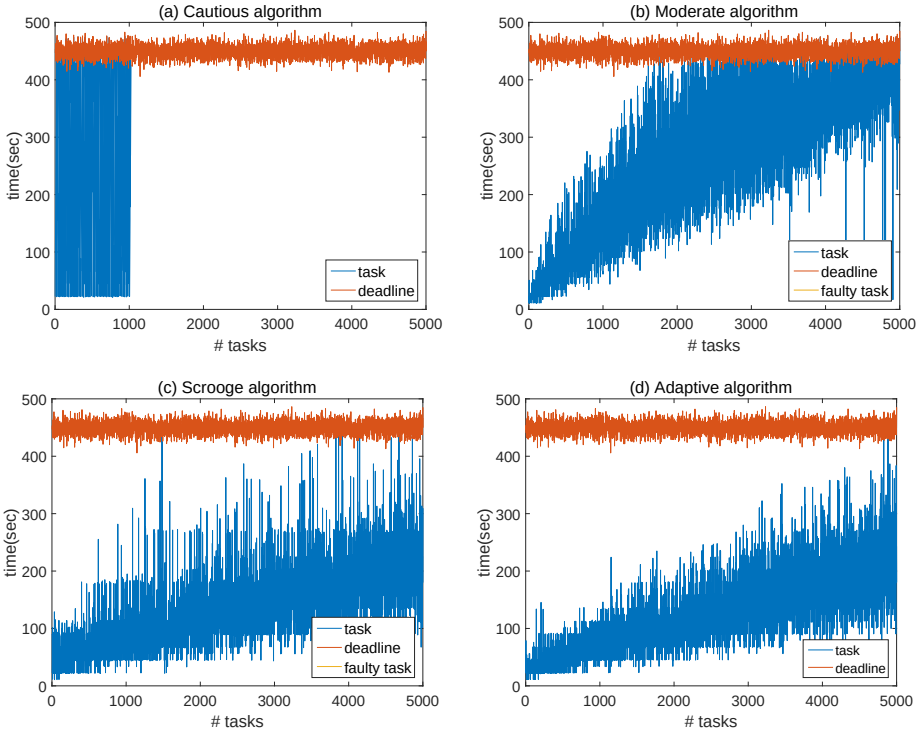

Figure 2: Computation time of tasks with low load

ity, while minimizing their spent budget. All protocols were evaluated by simulation under both low and high workloads.

Performance results show that in both workloads, Scrooge (resp., Cautious) is the cheapest (most expensive) and has the smallest (resp., the highest) task deadline fail ratio. Yet, it takes more (resp., less) time to schedule tasks since, due to the low (resp., high) reputation of the home (resp., trusted) nodes, a single task must be submitted to more (resp., just one) nodes. In contrast, in high workload, these metrics increase, and Scrooge can no longer ensure all tasks' deadlines.

In both workloads, all tasks are successfully scheduled by Adaptive and meet their deadline since Adaptive distributes the load among the cheapest nodes, but assigns tasks to more reliable nodes whenever this task cannot be executed in Home nodes. Further, its response time is slightly faster than with Scrooge. Such good results confirm the advantage of the adaptive protocol.

As future work, we intend to propose and evaluate protocols for the Bounded Budget scheduling optimizing problem, described in section 4.2. We also plan to investigate the impact of different reputation management strategies on the protocols' performance.

References

[1] A. Agbaria and R. Friedman. A Replication- and Checkpoint-Based Approach for Anomaly-Based Intrusion Detection and Recovery. In *IEEE Int. Conf. on Distributed Computing Systems Workshops*, pages 137–143, 2005.

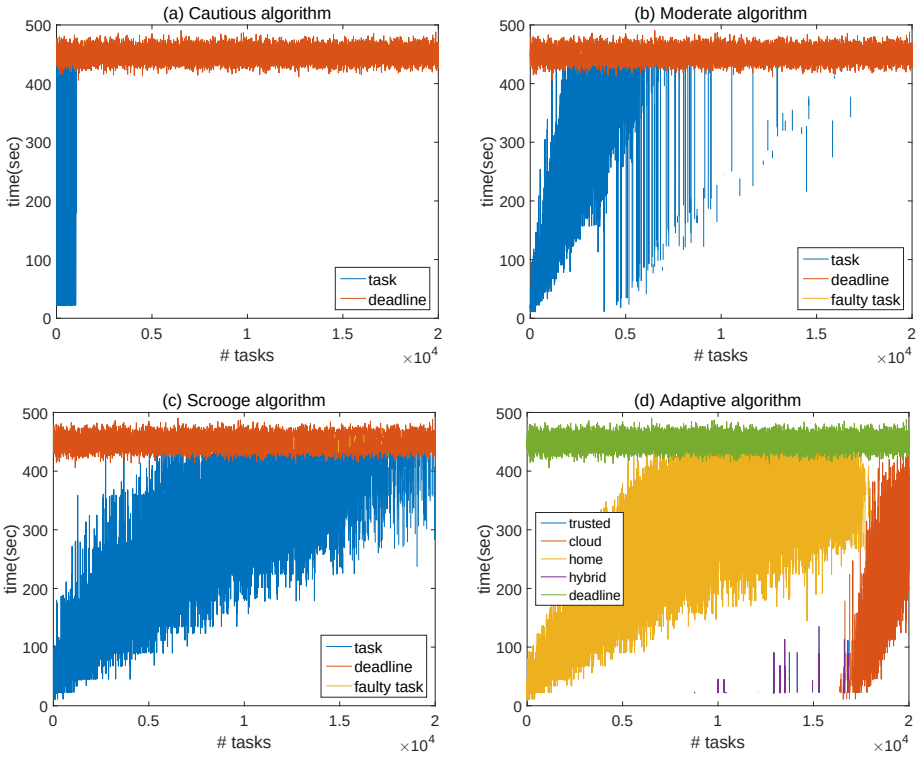

Figure 3: Computation time of tasks with high load

[2] D. P. Anderson. Boinc: A system for public-resource computing and storage. In *Proc. of the 5th IEEE/ACM Int. Workshop on Grid Computing*, pages 4–10, 2004.

[3] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. Seti@home: An experiment in public-resource computing. *C. ACM*, 45(11):56–61, 2002.

[4] L. Arantes, R. Friedman, O. Marin, and P. Sens. Probabilistic byzantine tolerance for cloud computing. In *Proc. of the IEEE SRDS*, 2015.

[5] J. Aspnes. Lower bounds for distributed coin-flipping and randomized consensus. *J. of the ACM*, 45:559–568, 1997.

[6] H. Attiya and K. Censor. Tight bounds for asynchronous randomized consensus. *J. ACM*, 55(5):20:1–20:26, 2008.

[7] H. Attiya and K. Censor-Hillel. Lower bounds for randomized consensus under a weak adversary. *SIAM J. Comput.*, 39(8):3885–3904, 2010.

[8] P.-L. Aublin, R. Guerraoui, N. Knežević, V. Quéma, and M. Vukolić. The next 700 bft protocols. *ACM Trans. Comput. Syst.*, 32(4):12:1–12:45, 2015.

[9] A. Bondavalli, S. Chiaradonna, F. Giandomenico, and J. Xu. An Adaptive Approach to Achieving Hardware and Software Fault Tolerance in a Distributed Computing Environment. *J. of Sys. Arch.*, 47(9):763–781, 2002.

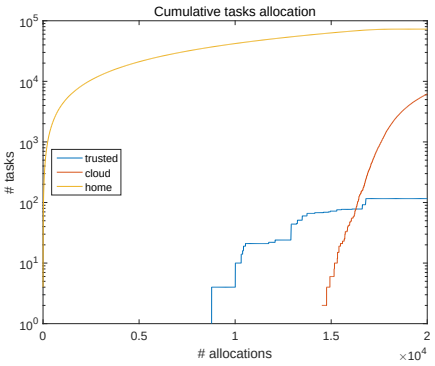

Figure 4: Task allocation using adaptive algorithm

[10] G. Bracha. An asynchronous [(n - 1)/3]-resilient consensus protocol. In *Proc. of the ACM PODC*, pages 154–162, 1984.

[11] Y. Brun, G. Edwards, J. Y. Bang, and N. Medvidovic. Smart Redundancy for Distributed Computation. In *Proc. of the 31st IEEE ICDCS*, pages 665–676, 2011.

[12] M. Castro and B. Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 20(4):398–461, 2002.

[13] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riche. Upright cluster services. In *Proc. of the ACM SIGOPS 22Nd SOSP*, pages 277–290, 2009.

[14] S. Delamare, G. Fedak, D. Kondo, and O. Lodygensky. Spequlos: a qos service for hybrid and elastic computing infrastructures. *Cluster Computing*, 17(1):79–100, 2014.

[15] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, 1988.

[16] R. Friedman, A. Mostefaoui, and M. Raynal. Simple and efficient oracle-based consensus protocols for asynchronous byzantine systems. *IEEE Trans. on Dependable and Secure Computing*, 2(1):46–56, 2005.

[17] R. Garcia, R. Rodrigues, and N. Preguiça. Efficient middleware for byzantine fault tolerant database replication. In *Proc. of the 6th ACM Conf. on Computer Systems*, pages 107–122, 2011.

[18] K. Hoffman, D. Zage, and C. Nita-Rotaru. A survey of attack and defense techniques for reputation systems. *ACM Comput. Surv.*, 42(1):1:1–1:31, Dec. 2009.

[19] Z. Hong and Z. Hai. Failure-aware resource scheduling policy for hybrid cloud. In *Int. Conf. on Computational Intelligence and Security*, pages 152–156, 2015.

[20] B. Javadi, J. Abawajy, and R. Buyya. Failure-aware resource provisioning for hybrid cloud infrastructure. *J. Parallel Distrib. Comput.*, 72(10):1318–1331, 2012.

[21] B. M. Kapron, D. Kempe, V. King, J. Saia, and V. Sanwalani. Fast asynchronous byzantine agreement and leader election with full information. *ACM Trans. Algorithms*, 6(4):68:1–68:28, 2010.

[22] M. Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Trans. Parallel Distrib. Syst.*, 12(10):1094–1104, Oct. 2001.

[23] M. Moca, C. Litan, G. C. Silaghi, and G. Fedak. Multi-criteria and satisfaction oriented scheduling for hybrid distributed computing infrastructures. *Future Generation Computer Systems*, 55:428–443, Feb. 2016.

[24] A. Mostefaoui, H. Moumen, and M. Raynal. Signature-free asynchronous byzantine consensus with $t < n/3$ and $o(n^2)$ messages. In *Proc. of the ACM SOSP*, pages 2–9, 2014.

[25] M. O. Rabin. Randomized byzantine generals. In *Proc. of the 24th Annual Symp. on Foundations of Computer Science (FOCS)*, pages 403–409, 1983.

[26] P. Resnick, K. Kuwabara, R. Zeckhauser, and E. Friedman. Reputation systems. *Commun. ACM*, 43(12):45–48, 2000.

[27] B. Rood and M. J. Lewis. Multi-state grid resource availability characterization. In *Proc. of the IEEE/ACM Int. Conf. on Grid Computing*, pages 42–49, 2007.

[28] L. Sarmenta. Sabotage-Tolerance Mechanisms for Volunteer Computing Systems. *Future Generation Computing Systems*, 14(4):561–572, 2002.

[29] J. D. Sonnek, M. Nathan, A. Chandra, and J. B. Weissman. Reputation-based scheduling on unreliable distributed infrastructures. In *26th IEEE ICDCS*, 2006.

[30] S. Toueg. Randomized byzantine agreements. In *Proc. of the ACM PODC*, pages 163–178, 1984.

[31] G. S. Veronese, M. Correia, A. N. Bessani, L. C. Lung, and P. Verissimo. Efficient byzantine fault-tolerance. *IEEE Trans. on Computers*, 62(1):16–30, 2013.

[32] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating Agreement from Execution for Byzantine Fault Tolerant Services. In *Proc. of the ACM SOSP*, pages 253–267, 2003.