# A Tree-based Protocol for Enforcing Quotas in Clouds

Ewnetu Bayuh Lakew*, Lei Xu†, Francisco Hernandez-Rodriguez*, Erik Elmroth*, Claus Pahl†
*Department of Computing Science, Umeå University, Sweden
{ewnetu, francisco, elmroth}@cs.umu.se
†School of Computing, Dublin City University, Ireland
{lxu,cpahl}@computing.dcu.ie

***Abstract*—Services are increasingly being hosted on cloud nodes to enhance their performance and increase their availability. The virtually unlimited availability of cloud resources enables service owners to consume resources without quantitative restrictions, paying only for what they use. To avoid cost overruns, resource consumption must be controlled and capped when necessary. We present a distributed tree-based protocol for managing quotas in clouds that minimizes communication overheads and reduces the time required to determine whether a quota has been exhausted. Experimental evaluation shows that our protocol reduces communication costs by 42% relative to a distributed baseline solution and is up to 15 times faster.**

***Keywords*-, Distributed Quota Monitoring, Distributed Quota Enforcement and Management, Distributed Credit Management, Clouds**

## I. Introduction

The growing popularity of cloud infrastructure has led to large number of services being hosted in the cloud. These services are complex software artifacts composed of many individual applications that are integrated to enable advanced functionalities such as social networks, e-commerce, or music streaming. To satisfy users' demands, services are deployed across geographical locations over hundreds or even thousands of nodes.

The virtually unlimited availability of resources in clouds enables them to be used as needed and without restrictions, following a pay-as-you-go pattern. While enticing, access to unlimited resources may be a liability in the absence of adequate control over resource usage. Careless use, misuse by end users (whether malicious or accidental), and programming errors can all cause service owners to go over budget. To avoid financial losses due to unintended consumption, service owners may choose to impose upper limits, i.e. *quotas*, on resource consumption.

Managing and enforcing quotas in clouds is challenging due to the large scale of the infrastructure, the potential number of end users accessing services, and the unpredictability (see [1]) of the ways in which services are used—for example, how often users arrive and how many resources they consume per visit. Accordingly, the key challenges for managing quotas in clouds are [2]:

1) To monitor services in real time, ensuring that resource usage across all nodes remains within specified global limits.
2) To satisfy resource demands immediately provided that the global quota is not exhausted.
3) To minimize network communication overheads.
4) To manage quotas with minimal intrusion and overhead to services.
5) To achieve scalability and fault tolerance.

Centralized approaches which verify that all resource consumption is within quota limits cannot be used due to the sheer scale of cloud infrastructures. Quota management techniques from distributed file systems [3] or grid computing [4] are also not suitable for clouds due to the unpredictability of services' resource demands. There are also distributed solutions in which a coordinator—which can be selected either statically [5] or dynamically [2] among nodes—performs global operations such as verifying that quotas have not been exceeded. While these distributed solutions (cf. [2] and [5]) address each of the challenges enumerated above to various extents, they require the coordinator to query every node before it can determine whether a quota has been exhausted. Querying every node causes latency and communication overheads as the number of nodes and the distances between them increase.

Here, we address these latency and communication overhead problems by proposing a tree-based protocol for enforcing quotas in clouds. In our protocol, all nodes running a service maintain a local quota allocation so that permissions can be given immediately provided that the local allocation is not exhausted. When local quota allocations are exhausted, nodes request extra quota only from nearby neighbors instead of polling all nodes (see Section IV). Our protocol has negligible computational overhead because the quotas are set using a simple exponential moving average (EMA) (see Section III) and communication is performed only with close neighbors.

Our solution is highly available and scalable because each branch of the tree can set quotas for its children and several of these operations can be performed concurrently. In practice, quota allocations are dynamically transfered from idle nodes to active nodes without affecting normal service operation. Experimental evaluations showed that our protocol reduces communication costs by 42% relative to our previous distributed solution [2] and is up to 15 times faster (see

Section V).

## II. Related Work

The problem of online, distributed quota management and enforcement has been addressed by several distributed storage and network systems. A number of works have focused on how to monitor a global count of resources used at different nodes. Their aim was to ensure communication efficiency and guarantee correctness while continuously monitoring a specific parameter at each node and reporting violations to a coordinator node as they occur. The coordinator processes the collected monitoring data for different purposes such as finding top-$k$ items [9], calculating sums and counts [10], or detecting violations over distributed data streams with guaranteed error bounds [11], [12]. However, these works focus on soft limit enforcement where temporary violations of quota limits are tolerated. Moreover, these solutions employ a single coordinator that is selected *a priori*, which limits their scalability.

Our work is similar to that of Raghavan et al. [1], who focused on controlling the network bandwidth consumed by services in a cloud. In their method, requests at a local site are served as long as the requested bandwidth is less than the local threshold. Temporary bursts are dropped even if the global limit is not surpassed, which may be intolerable from a service point of view. In contrast, our approach does not drop temporary bursts because quotas are monitored and adjusted on demand as long as the global limit is not exceeded.

Relevant work has also been conducted on hard limit enforcement in storage and grid environments [4], [3]. Karmon et al. [4] presented a solution that utilizes a tree structure to efficiently distribute and enforce quotas in grid environments. However, their work does not address key issues such as determining surpluses and demands. Moreover, their approach denies requests if the local quota allocation is lower than the amount requested even if the global limit has not been exhausted. The solution of Pollack et al. [3] is similar in some respects but introduces the use of vouchers with defined periods of validity that are given to nodes. A node's allocated quota is taken away if it is not used during its period of validity. However, the control and assignment of vouchers is done using a single quota server whereas our work operates in a decentralized manner. The work in [2] also addresses distributed quota enforcement in a multi-cluster environment by querying every cluster during quota distribution. However, querying every node causes latency and communication overheads as the number of nodes and the distances between them increase.

Various researchers have investigated prepaid accounting frameworks and protocols [13] [14] [15] [16]. While these works provide flexible solutions for charging for service usage, their accounting systems are centralized. This may generate a bottleneck in cases where large numbers of services are consuming resources simultaneously.

## III. Solution Overview

We investigate ways of managing quotas (governing resources such as prepaid credit, CPU hours, CPUs, RAM allocation, storage space, and network resources) consumed by customers' services running on multiple nodes in a cloud infrastructure. We assume that each customer's service, running on a physical node, is associated with a quota daemon. The quota daemon is a lightweight process that manages the node's local quota and coordinates with other quota daemons belonging to the same customer on other physical nodes when the local quota is not sufficient to satisfy local demand. When the local quota budget is below or close to a predefined threshold, the local quota daemon gathers additional quota allocations from nearby nodes and redistributes them based on the rates of service consumption on each node.

This section presents the system model and a high-level view of our solution.

### A. System Model

We assume that a given customer (tenant) has services running on several physical nodes. In addition, a single physical node may host services belonging to multiple tenants, each of which is isolated using a virtual machine (VM). The goal of quota management is to make sure that the sum of the quotas consumed concurrently by an individual tenant over all of the physical nodes running their services does not exceed that tenant's global limit. A decentralized approach in which a quota daemon is attached to each tenant's service on each physical node is used to manage their quota consumption. Figure 1 shows the architecture of the quota daemon running on a single physical node. The quota daemon interacts with other quota daemons running on other physical nodes and communicates information about the availability of free quota units with other physical nodes running services belonging to the same tenant. It ensures that the service running on any given node does not consume resources at a level that would cause the tenant to exceed their global quota.

We assume that the physical nodes running the tenant's services are connected using a tree topology that is constructed and maintained using distributed spanning tree algorithms such as that described by Garg et al. [17]. Thus, if the local quota on a given node becomes insufficient to meet the tenant's local resource demands, the local quota daemon interacts recursively with its neighboring nodes[1] in the tree to identify those with unused quota allocations. These "free" quota allocations are then transferred to the overloaded node until its demand is satisfied.

### B. High Level View of the Solution

Quota enforcement policies can be roughly divided into two categories: *soft limits* [18], [5], [19] and *hard limits* [2]. *Soft limit* quota enforcement ensures that resource consumption does not exceed a predefined global limit for an extended

[1]We use the term *node* to denote the physical machine running a particular tenant's service and its associated quota daemon. The terms *node* and *quota daemon* are therefore used interchangeably henceforth.

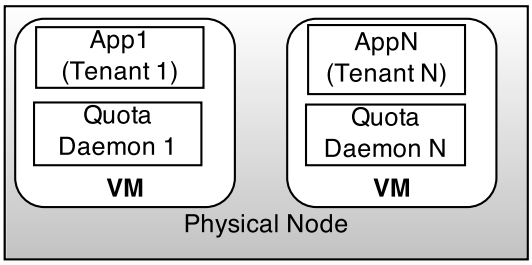


Figure 1: Architecture of the quota system on an individual physical node. Quota requests issued by applications belonging to different tenants are handled by different Quota Daemons. If a tenant's local resource demands exceed their local quota allocation, their Quota Daemon communicates with other nodes running the tenant's applications to identify unused quota allocations elsewhere that can be used to satisfy this demand.

Table I: Notation used throughout the paper

| | |
|---|---|
| $n_k$ | an arbitrary node on which a quota daemon is hosted |
| $n_k \cdot child$, $n_k \cdot parent$ | the child and parent of $n_k$, respectively. |
| $n_{root}$ | the outermost node or the root of the tree structure, where $n_{root} \cdot parent = \emptyset$. If $n_k \cdot child = \emptyset$, then $n_k$ is a leaf of the tree. |
| $y_k$ | a set of nodes whose parent is $n_k$, where $\forall n_a \in y_k$, $n_a \cdot parent = n_k$. |
| $\|y_k\|$ | the number of nodes constituting the set $y_k$. |
| $q_k$ | the number of quota units available at $n_k$ in any particular instance. |
| $Q^0$ | the tenant's initial global quota |
| $Q_k$ | the sum of the number of quota units available to children of $n_k$ in any particular instance. |
| $\Delta t_c(kh)$ | the time interval required to calculate the availability of quota units and distribute them from $n_k$ to $n_h$. |
| $\Delta t_d(kh)$ | the time interval required for $n_k$ to receive a reply from $n_h$. |
| $\tau_k$ | a threshold below which quota re-allocation is triggered. |
| $\theta_k$ | the time interval over which $\tau_k$ is expected to be valid during re-allocation by the node $n_k$. |
| $Q_{ak}$ | the aggregate quota allocation available at $n_k$ in any particular instance. |
| $T_{ak}$ | the aggregate threshold value known to $n_k$ in any particular instance during the aggregation and distribution phases. |

period of time but tolerates intermittent or temporary violations of the global limit; it is typically used in applications such as network traffic control or DDoS attack control. On the other hand, *hard limit* quota enforcement ensures that resource consumption never exceeds the global limit. Hard limit enforcement is applicable when regulating the consumption of resources such as prepaid credit, CPU hours, number of CPUs, RAM allocation, and storage space. This work focuses on hard limit quota enforcement.

When a service is initiated, the user's global quota allocation may be distributed randomly over all nodes within the spanning tree. Once a node $n_k$ exhausts its local quota (i.e. its local quota falls below or gets close to the threshold $\tau_k$), it triggers a re-allocation event and requests the allocation of additional quota units from either its parent (if the node is a leaf or the total number of quota units available from its descendants is not sufficient) or its descendants (see Section IV). The threshold $\tau_k$ for node $n_k$ is equal to the maximum number of quota units that are expected to be consumed by the tenant's service that is running on that node under normal conditions. A given node $n_k$ will therefore reserve $\tau_k$ quota units for its own use before making any surplus units available to its parent. This is done to avoid service suspension during quota redistribution. The value of $\tau_k$ is dependent on the rate at which quota units are consumed by $n_k$ and the time interval required for the transfer of additional quota units. Therefore, $\tau_k$ is estimated on the basis of a time interval that is denoted by $\theta_k$ and whose value is calculated as follows:

$$\theta_k = \Delta t_c\,(kh) + \Delta t_d\,(kh) \tag{1}$$

$\Delta t_c(kh)$ and $\Delta t_d(kh)$ can be estimated using recent historical data and adjusted accordingly.

Let $t_s$ and $t_e$ denote the start and end of a time interval within which the quota on node $n_k$ is consumed continuously, where $t_e = t_s + \theta_k$. Moreover, let $f(n_k, t)$ denote the consumption rate of the service hosted on $n_k$, which may vary with consumption time (denoted by $t$). Based on the above, $\tau_k$ can be estimated as follows:

$$\tau_k = \int_{t_s}^{t_e} f\,(n_k, t)\,dt \tag{2}$$

When $n_k$ receives an allocation request (either from its parent or one of its children), it will collect allocatable quota units from its children. When each child $n_h$ receives a request from its parent $n_k$[2], it first reserves $\tau_h$ quota units that it expects to consume over the time interval $\theta_k$. Each child $n_h$ then reports its consumption rate and the number of quota units that it has available for reallocation (i.e. its total number of quota units minus the number of reserved units) to $n_k$.

Assuming that node $n_k$ is coordinating the re-allocation process, it will calculate the total number of allocable quota units $Q_k$ available from its children and itself as:

$$Q_k = q_k + \sum_{j=1}^{|y_k|} q(h), where\ q(h) = \begin{cases} 0 & q_h - \tau_h \leq 0 \\ q_h - \tau_h & q_h - \tau_h > 0 \end{cases} \tag{3}$$

The quota units are dispensed dynamically based on each node's demand, enabling the system to react rapidly to changes in demand in a local manner. Demand is likely to be heaviest in nodes where it has previously been heavy. Therefore, over time, quota units are transferred to nodes with high demand. Specifically, quota units are allocated across nodes based on each node's reported rate of consumption. This is done on the assumption that a node will continue consuming quota units at the same rate as it has in the past. We use exponential moving averages (EMA) [2], [1] to calculate consumption rates for

[2]This operation is a recursive process. That is, when collecting quotas the request is forwarded to the leaf nodes while during the reply process, the quota is aggregated at each level by the parents until it reaches the destination node, which would be $n_k$ or $n_k$'s parent in this case.

individual nodes because they enable fair allocation, smooth out transient peaks in demand, and reduce lag by applying greater weightings to more recent rates.

Let $r_k^p$ denote the consumption rate on $n_k$ after aggregation event $p-1$ where $p > 1$ or after the first aggregation where $p = 1$. The quota consumption rate $r_k^p$ can be calculated for any node $n_k$ as:

$$r_k^p = \begin{cases} \left(q_i^0 - q_k^1\right)/Q^0 & p = 1 \\ (1-\alpha)r_k^{p-1} + \left(\alpha(q_k^{p-1} - q_k^p\right))/Q^0 & p > 1 \end{cases} \quad (4)$$

The coefficient $\alpha$ in Equation (4) represents the degree by which a given rate's weighting is reduced. We chose an $\alpha$ value of 0.50 in our experiment because preliminary evaluations showed that this provided effective smoothing of transient spikes.

The number of quota units $q_k$ allocated to the parent $n_k$ can be calculated as:

$$q_k = (Q_k) \times \frac{r_k^p}{r_k^p + \sum_{j=1}^{|y_k|} r_j^p} \quad (5)$$

The number of resource units receieved by the child $n_a$ after re-allocation is denoted by $q_a$ and can be calculated as:

$$q_a = (Q_k) \times \frac{r_a^p}{r_k^p + \sum_{j=1}^{|y_k|} r_j^p} \quad (6)$$

## IV. The Protocol

This section presents an overview of the protocol and the method used to aggregate and distribute quota units across the participating nodes.

### A. Overview Of The Protocol

The quota management protocol is highly scalable and distributed cloud-wide. It uses a tree-based overlay network to exchange free quota units with other nodes. We assume that the nodes in the system are completely connected, forming a tree-based topology over a real network (e.g., [17]). The global quota is initially distributed randomly over the tree's nodes. The goal is that the system should operate locally, giving a node immediate permission to use its own free quota units when its current level of demand is well below its total quota allocation. However, in the event that one or more of a node's children needs more quota units than it has available locally, every parent in the tree can act as a coordinator of quota redistribution by identifying its own 'spare' quota units and those on other (idle) child nodes, and reallocating them to the child with high demand. If the parent, after collecting free quota units from its children, does not have enough to satisfy the child's demand, it can in turn request quota from its parent. This process continues recursively until it reaches the root of the tree provided that the upper bound is not reached.

Each coordinating parent performs operations such as collecting extra quota units from child nodes, executing quota re-allocation, and redistributing the new quota units to child nodes that are participating in the re-allocation process. This allows multiple independent quota re-allocation decisions to be executed simultaneously within different subtrees having non-overlapping nodes.

The protocol has two phases: *aggregation* and *distribution*. When triggered, the protocol first performs *aggregation* of surplus quota units and then *distribution* of quota units to nodes where they are needed. Information on surplus quota units and the thresholds of each participating node across the spanning tree is aggregated incrementally via a series of local computations during the *aggregation phase*. Each parent computes partial aggregate data on the thresholds and available quota units within its subtree by combining the partial aggregate data for its children's quota units and thresholds with its own free quota and threshold values respectively. For a node $n_k$, the partial quota aggregate $R_{ak}$ and the partial threshold aggregate are computed using Equations 7 and 8, respectively.

$$Q_{ak} = Q_k + \sum_{j=1}^{|y_b|} Q_{aj} \quad (7)$$

$$T_{ak} = \tau_k + \sum_{j=1}^{|y_b|} T_{aj} \quad (8)$$

The aggregation operation terminates when the aggregated value is greater than the aggregated thresholds of its children and its own local threshold multiplied by a constant (i.e. when $Q_{ak} > \beta T_{ak}$, where, $\beta = 1, 2, ..., n$) or when it reaches the root of the entire tree. The parameter $\beta$ is a constant that specifies how many quota units should be aggregated for distribution. Increasing the value of $\beta$ increases the number of aggregated quota units available for distribution, which may avoid the need for the immediate invocation of the protocol since nodes with high demand will receive more quota units to begin with. As a result, high values of $\beta$ can reduce communication overheads and improve performance. In our experiment, we set $\beta$ to 1.

The aggregation phase can be triggered by any node if its local quota allocation falls below or becomes close to its threshold. If it is triggered by a leaf node, the leaf node sends a request to its parent asking for more quota units. The parent then collects quota surpluses and thresholds from all of its children. To facilitate the process for non-leaf children the parent also transfers its own surplus quota units and threshold. Each non-leaf child can in turn recursively collect quota surpluses and thresholds from their children if the surplus received from parent plus its local surplus is less than the product of $\beta$ and the combined thresholds of the node itself and its parent (surpluses are calculated using Equation 3). After collecting data from all of its descendant nodes, the parent node will pass on the aggregated data to its own parent if $Q_{ak} \leq \beta T_{ak}$, at which point the process will be repeated by the parent's parent. In the worst case, i.e. if $Q_{ak} \leq \beta T_{ak}$ remains true after the process has been performed by the parent's parent, the aggregated quota and thresholds will be

passed further down the tree until they eventually reach the root, at which point the process will terminate. This situation will only occur if the tenant's overall quota allocation is very low and may indicate that the service should be terminated unless the tenant's quota can be replenished or increased.

When the protocol is triggered by any non-leaf node, the parent collects surplus data from all of its children instead of asking its parent. The operations disucssed above are then performed. Section IV-B presents the algorithm used in the aggregation phase.

Once the *aggregation phase* is complete, quotas are distributed to all of the nodes that participated in the aggregation phase during what is termed the *distribution phase*. The distribution phase is based on another recursive process in which new quota units are distributed from the parent, where the aggregation phase terminated, to each node in the subtree that participated in the aggregation phase. This is done according to Equations 5 and 6. Note that the quota consumption rate $r_j^p$ of child node $j$ in Equation 5 may represent an aggregated quota consumption rate (i.e. the child's local consumption rate plus that of its descendants if those descendants participated during aggregation phase). Section IV-B2 presents the algorithm used in the distribution phase.

### B. The Algorithm

As stated previously, a quota re-allocation event is triggered when the local quota balance of a node $n_k$ is below or close to its threshold $\tau_k$. The first phase of quota re-allocation is *aggregation*, where extra quota units are collected from descendant nodes (and from ancestors if sufficient quota units cannot be collected from the descendants). During the subsequent *distribution phase*, the collected 'extra' quota units are then redistributed to each participating node based on their rates of consumption. Algorithms 1 and 2 outline the processes used in the aggregation and distribution phases, respectively.

*1) Aggregation phase:* The aggregation phase is a recursive process that aggregates 'spare' quota units for redistribution over a subtree. Any node $n_k$ that is not participating in another aggregation process (i.e, $n_k$.PARTICIPATING=0) can trigger aggregation if $q_k \leq \tau_k$ (depicted on lines 5–8). Any node receiving an aggregation request will in turn perform the aggregation process (depicted on lines 9–19).

The procedure *AGGREGATE()* (lines 22–46) governs what happens during the aggregation phase. First it checks whether the node is already participating in the process; if so, it returns *PARTICIPATING* (lines 23–24). Otherwise, the node changes its state to participating and calculates the numbers of local quota units to reserve and 'free' local quota units that can be returned to the parent (lines 26–28). The next action depends on who initiated the aggregation. If the node itself initiated the aggregation event or was signalled to do so by its child, it updates the aggregated quota and threshold values and calls the *AGGREGATE()* procedure in its children (lines 32–35). On the other hand, if the aggregation request was received from a parent node, the receiving node updates the aggregate quota and threshold. If the number of aggregated free quota units exceeds the product of the aggregated threshold and $\beta$ then the aggregated quota, aggregated threshold and aggregate average consumption rate are returned. Otherwise, the aggregation request is sent to the node's children. This process is shown on lines 36–44.

**Algorithm 1** Aggregation algorithm.

```
1:  n_k.PARTICIPATING=0
2:  n_k.Q_ak=0
3:  n_k.T_ak=0
4:  loop
5:      if q_k < τ_k && n_k · child = ø && n_k.PARTICIPATING=0 then
6:          Send request to n_k · parent for more quota
7:      else if q_k < τ_k && n_k · child ≠ ø && n_k.PARTICIPATING=0 then
8:          n_k · parent· AGGREGATE(n_k,ø)
9:          if n_k.Q_ak ≤ βn_k.T_ak && n_k ≠ root then
10:             n_k · parent· AGGREGATE(n_k · parent,ø)
11:         end if
12:     else if Received request for more quota from child nodes then
13:         AGGREGATE(n_k,ø)
14:         if n_k.Q_ak ≤ βn_k.T_ak && q_k ≠ root then
15:             AGGREGATE(n_k · parent,ø)
16:         end if
17:     else if Received for aggregate from parent node n_k · parent then
18:         AGGREGATE(n_k, n_k · parent)
19:     end if
20:     DISTRIBUTE( n_p)
21: end loop

22: function AGGREGATE(n_k, n_p)
23:     if n_k.PARTICIPATING=1 then
24:         return PARTICIPATING
25:     end if
26:     n_k.PARTICIPATING=1
27:     estimate localThreshod τ_k using Equation 2
28:     temp=n_k.localQouta − n_k.localThreshold
29:     if temp<0 then
30:         temp=0
31:     end if
32:     if n_p =ø then
33:         n_k.Q_ak = temp
34:         n_k.T_ak = n_k.localThreshold
35:         sendToChild(n_k)
36:     else if n_p ≠ ø then
37:         n_k.Q_ak = n_p.Q_ap + temp
38:         n_k.T_ak = n_p.T_ap + n_k.localThreshold
39:         if n_k.Q_ak > βn_k.T_ak OR n_k is leaf node then
40:             store the consumption rate and that of each child locally
41:             return n_k.Q_ak, n_k.T_ak and consumption rate
42:         else
43:             sendToChild(n_k)
44:         end if
45:     end if
46: end function

47: function s(e)ndToChild(n_p)
48:     for each child n_c do                    ▷ call child aggregate method
49:         n_c.AGGREGATE( n_c, n_p)
50:     end for
51: end function
```

*2) Distribution phase:* The distribution phase is the reverse of the aggregation phase: it distributes quota units to all nodes that participated in the aggregation phase. Algorithm 2 presents the operations performed during distribution phase. Each node in the hierarchy that participated during the aggregation phase recursively iterates through its children and performs redistribution operations. This is shown in lines 2–10.

**Algorithm 2** Distribution Algorithm.

```
1:  function DISTRIBUTE( n_k)
2:      Update n_k · localQuota using Equation 6
3:      Adjust n_k · localThreshold
4:      for each child n_c do
5:          if n_c · PARTICIPATING = 1 then
6:              Calculate n_c.localQuota using Equation 5
7:              n_c·DISTRIBUTE( n_c)
8:          end if
9:      end for
10:     n_k · PARTICIPATING = 0
11: end function
```

*3) Complexity of the Algorithm:* The complexity of the algorithm in terms of the number of messages sent depends on the number of children (branching factor) and the number of participating nodes during the two phases of the protocol. Let $N$ denote the total number of nodes and $M$ the number of children for a non-leaf node in the spanning tree. Three messages are exchanged between each child and the parent: two during the aggregation phase (one to request extra quota units from the parent and another conveying the child's response to the request) and one during the distribution phase to distribute quota units from parent to child. As a result, the total number of messages exchanged to complete the protocol is between $3M$ and $3N$.

### C. Fault Tolerance of the Protocol

There are different kinds of failures that would affect the protocol. To structure our discussion, these failures are categorized into three scenarios. *1)* Link disconnection or node failure during protocol execution, which may result in quotas being deducted from the source's quota allocation but not added to the target (if the failure occurs during the aggregation phase) or quota units being distributed but not received by the target daemon (if the failure occurs during the distribution phase). *2)* A node crashes, taking with it any quota units in its possession. *3)* Network partitioning, which would result in a set of different independent and disconnected subtrees.

We use different approaches to tackle these failures. Every node keeps a record of the total number of quota units, unused and used, to overcome the first failure mode. Once the nodes are reconnected, the aggregation phase is re-run from the root to collect all used and unused quota units from all nodes. The root node then computes the sum of used and unused quota units and compares the resulting value to the initial global quota. The excess of the initial global quota over the computed sum is the number of lost quota units. No special action is required to handle the second and the third type of failures. In the second case, the node can resume normal activity after rejoining the network. In the third case, the protocol resumes as normal after network reconnection.

## V. Experimental Evaluation

### A. Experiment Settings

We evaluated our protocol by emulating quota enforcement for a distributed storage service. Our prototype was implemented using Java RMI. The testbed consisted of 20 hosts. Thirteen of the hosts were equipped with AMD Opteron(TM) Processors, with each host having 32 2.1GHz CPUs and 64GB RAM. A further 7 hosts were each equipped with 1.8GHz quad-core AMD Opteron(TM) CPUs and 4GB RAM. All hosts were connected via our campus intranet, which is based on Gigabit Ethernet. Each host ran up to seven instances (nodes) of the prototype to support the simulation of larger networks. We studied systems of up to 100 nodes with each node having at most 5 neighbors (children).

The response time and communication efficiency of our protocol were compared to those for our previous distributed solution [2] in which each node ran a quota manager. Each quota manager served storage requests provided that its local quota allocation was sufficient (i.e. above a predefined threshold). If the requested quota was greater than the local node balance, the quota manager triggered an allocation event and all other nodes sent their extra balance and the number of quota units they had consumed since the previous allocation event. Based on these consumption rates, the quota manager that triggered the event then calculated the new allocations for each node and redistributed the available quota units.

We used two datasets to generate workloads. One dataset, *WorldCup*, consisted of real-world traces of HTTP requests across a set of distributed web servers. The trace data was obtained from the organizers of the 1998 FIFA Soccer World Cup [20], who maintained a popular web site that was accessed over 1 billion times between April 30, 1998 and July 26, 1998 and was served by 30 servers distributed across different geographic locations around the world. Our experiments were conducted using server log data consisting of 57 million page requests distributed across servers that were active during that period. We replayed the trace in such a way that the server identifier was used to indicate the node that received the request and the size of data requested was used in place of the number of quota units requested by the storage service.

The other dataset, *Synthetic*, contained randomly generated traces that gave us the freedom to evaluate parameters that were not controlled in the real world trace. For instance, we were able to increase the number of nodes in order to evaluate scalability and to increase the number of requests to understand the behavior of our solution as the number of requests increased. We also varied the size of the capacity requested in order to study how this affected the communication overhead. The synthetic trace distributed requests to nodes according to a Zipf distribution. We employed this distribution because it has been conjectured that the Internet [21] is characterized by distributions of this type, and because it allowed us to vary the number of requests received by each node.

We ran the experiment until the global quota was exhausted. The results presented herein are based on average values from five different executions. The communication cost was measured in terms of the number of messages exchanged between different nodes for quota reallocation. For example, a message sent or received from a parent to a child had a cost of one while messages from a grand parent or a grand child had a cost of two.

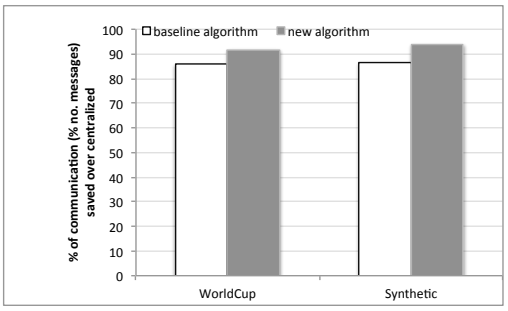


Figure 2: Communication cost reductions achieved using the baseline and new algorithms relative to a centralized approach. The cost reductions are expressed as percentages indicating the ratio of the total number of messages exchanged during reallocation for the tested algorithms to that for the centralized protocol.

### B. Results

Figure 2 shows the communication cost reduction achieved using our previous algorithm [2] and the protocol reported herein as percentages of the communication costs incurred when using a centralized approach for both the WorldCup and Synthetic datasets. We refer to our previous approach [2] as the *baseline algorithm* and the protocol presented herein as the *new algorithm* in the remainder of this paper. For both datasets, the baseline algorithm reduces communication costs by around 85% while the new algorithm achieves reductions of around 92%. The new algorithm thus generates less traffic than the baseline algorithm. This is because under the new algorithm, nodes that need extra quota units send quota requests out progressively, initially focusing on their immediate neighbors rather than sending messages out to all nodes immediately.

Table II shows the two algorithms' average response times for quota re-allocation with the tested datasets. For both datasets, the new algorithm has a faster response time than the baseline algorithm. This is because most of the time the new algorithm interacts only with closer neighbors than those contacted when using the baseline algorithm.

The table also shows that the response time for the WorldCup dataset was greater than that for the synthetic dataset. This is because a load balancer was used to distribute requests across nodes when the WorldCup dataset was generated, so the trace in this dataset contains requests which are almost uniformly distributed across the whole workload. Consequently, each node's consumption of quota units is comparatively uniform and all nodes have a similar and high probability of triggering quota re-allocation. This creates appreciable performance overhead, which increases response times. In contrast, the synthetic dataset was created using a Zipf distribution that spreads requests across nodes. Consequently, most of the quota consumption in this case occurs in a small number of nodes, resulting in a shorter response time.

Figure 3 shows the relationship between the relative communication cost and the proportion of global quota remaining for the WorldCup dataset (we omitted the results for synthetic trace because they did not differ greatly). It is clear that the communication cost is very low to begin with. This is because each node can meet its demand using its local quota. However, as the local quota becomes insufficient to satisfy the local demand in some nodes, these nodes start interacting with their neighbors, causing communication costs to start rising. The communication cost becomes very high when the amount of residual global quota is less than 10% of the initial value because nodes immediately exhaust their allocation and demand more. This is especially true for the new algorithm because nodes cannot satisfy their demands by obtaining quota units from near neighbors and therefore end up communicating extensively with very distant nodes. In general, nodes communicate less when the total allocation is high. However, as the total quota alocation declines, re-allocation requests become more frequent and communicaton costs increase.

Table II: Average response time in ms.

| Algorithm | Synthetic trace | WorldCup trace |
|---|---|---|
| Baseline algoritm | 69.5 | 75.8 |
| New algoritm | 13 | 22.7 |

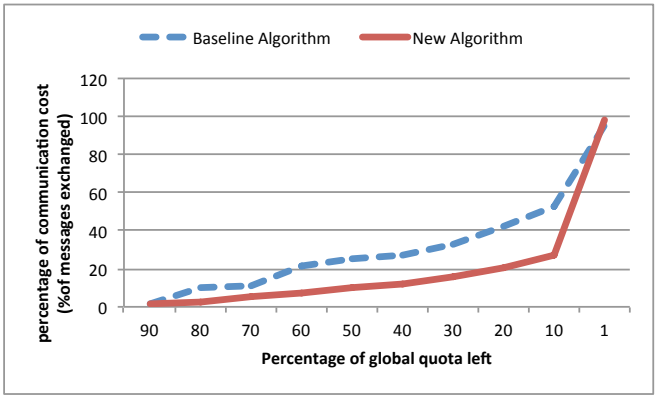


Figure 3: Relative communication costs as a function of the remaining global quota allocation.

To investigate the scalability of the two algorithms, we performed a number of tests with different numbers of nodes that were run until the global quota allocation was exhausted. These tests were performed using the Synthetic workload, with the same initial global quota allocation in each case. Communication costs were measured in terms of the total number of messages exchanged and average response time for each test. The results obtained are shown in figures 4 and 5, respectively. Figure 4 shows that as the number of nodes increases, the new algorithm becomes more efficient at reducing communication overheads than the baseline algorithm. This is because under the new algorithm, nodes generally communicate only with their near neighbors. The new algorithm therefore has a better average response time than the baseline algorithm. For example, as shown in Figure 5, when the number of nodes was increased from 10 to 100, the average response time increased from 6.35ms to 25ms for the new algorithm but went from 19.74ms to 346ms for the

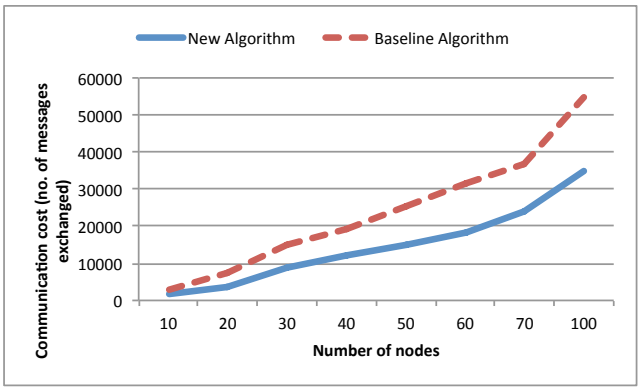


Figure 4: Effect of increasing numbers of nodes on the number of messages exchanged.

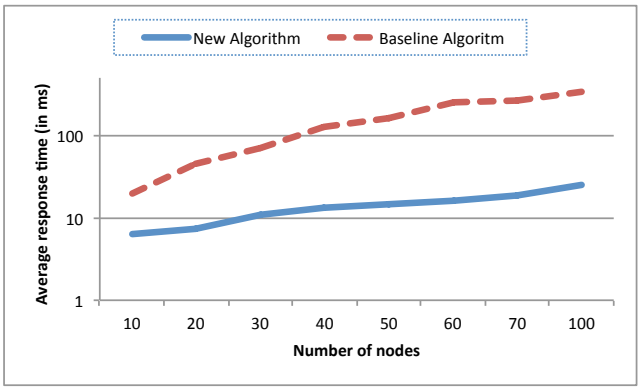


Figure 5: Effect of increasing numbers of nodes on response times.

baseline algorithm.

## VI. Conclusion

We present a hierarchical quota enforcement protocol for services running on multiple nodes in a cloud environment. Our goal was to ensure that the aggregated quota consumption across nodes does not exceed a global limit. As part of the quota management process, unused quota allocations are transferred to nodes where they are in demand rather than being wasted on idle nodes. Global polls are triggered only when the local quota allocation at a node falls below a defined threshold. When this happens, quota allocations are redistributed across all polled nodes based on their rates of resource consumption in a manner that minmizes the need for further global polls in the short term.

We compared our solution to centralized and distributed approaches. Our solution produces less communication overhead and has faster response times when performing quota redistribution among demanding nodes. Moreover, experimental results indicate that our solution is superior to the tested alternatives with respect to various non-functional requirements: it offers better performance and is stronger in terms of concurrency, scalability, availability, and fault tolerance.

## References

[1] B. Raghavan, K. Vishwanath, S. Ramabhadran, K. Yocum, and A. C. Snoeren, "Cloud control with distributed rate limiting," in *Proceedings of the 2007 conference on Applications, technologies, architectures, and protocols for computer communications*, ser. SIGCOMM '07. New York, NY, USA: ACM, 2007, pp. 337–348.

[2] E. B. Lakew, F. Hernández-Rodriguez, L. Xu, and E. Elmroth, "Management of distributed resource allocations in multi-cluster environments," in *the IEEE 31st International Performance Computing and Communications Conference (Austin, TX, USA, 2012), IPCCC'12*, 2012, pp. 275–284.

[3] K. Pollack, D. Long, R. Golding, R. Becker-Szendy, and B. Reed, "Quota enforcement for high-performance distributed storage systems," in *Mass Storage Systems and Technologies, 2007. MSST 2007. 24th IEEE Conference on*, Sept 2007, pp. 72–86.

[4] K. Karmon, L. Liss, and A. Schuster, "GWiQ-P: an efficient decentralized grid-wide quota enforcement protocol," *14th IEEE International Symposium on High Performance Distributed Computing, 2005. HPDC-14. Proceedings.*, pp. 222 – 232, july 2005.

[5] S. Meng, S. R. Kashyap, C. Venkatramani, and L. Liu, "Remo: Resource-aware application state monitoring for large-scale distributed systems," in *Proceedings of the 2009 29th IEEE International Conference on Distributed Computing Systems*, ser. ICDCS '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 248–255.

[6] A. Benoit, R. G. Melhem, P. Renaud-Goud, and Y. Robert, "Assessing the performance of energy-aware mappings." *Parallel Processing Letters*, vol. 23, no. 2, 2013.

[7] B. D. Young, J. Apodaca, L. D. Briceno, J. Smith, S. Pasricha, A. A. Maciejewski, H. J. Siegel, B. Khemka, S. Bahirat, A. Ramirez, and Y. Zou, "Energy-constrained dynamic resource allocation in a heterogeneous computing environment." in *ICPP Workshops*. IEEE, 2011, pp. 298–307.

[8] M. Marzolla and R. Mirandola, "Dynamic power management for qos-aware applications," *Sustainable Computing: Informatics and Systems*, vol. 3, no. 4, pp. 231 – 248, 2013.

[9] B. Babcock and C. Olston, "Distributed top-k monitoring," in *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '03, 2003, pp. 28–39.

[10] R. Keralapura, G. Cormode, and J. Ramamirtham, "Communication-efficient distributed monitoring of thresholded counts," in *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '06, 2006, pp. 289–300.

[11] S. Agrawal, S. Deb, K. V. M. Naidu, and R. Rastogi, "Efficient detection of distributed constraint violations," in *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*, April 2007, pp. 1320–1324.

[12] S. Meng, S. R. Kashyap, C. Venkatramani, and L. Liu, in *Proceedings of the 2009 29th IEEE International Conference on Distributed Computing Systems*, 2009, pp. 248–255.

[13] B. Jennings and P. Malone, "Flexible charging for multi-provider composed services using a federated, two-phase rating process," in *NOMS*, 2006, pp. 13–23.

[14] F. Bormann, S. Flake, J. Tacken, and C. Zoth, "Third-party-initiated context-aware real-time charging and billing on an open soa platform." in *22nd International Conference on Advanced Information Networking and Applications Workshops*, 2008, pp. 1375–1380.

[15] P. Gardfjäll, E. Elmroth, L. Johnsson, O. Mulmo, and T. Sandholm, "Scalable Grid-wide capacity allocation with the SweGrid Accounting System (SGAS)." *Concurrency and Computation: Practice and Experience*, vol. 20, pp. 2089–2122, 2008.

[16] E. Elmroth, F. G. Marquez, D. Henriksson, and D. P. Ferrera, "Accounting and billing for federated cloud infrastructures," in *Proceedings of the 2009 Eighth International Conference on Grid and Cooperative Computing*, ser. GCC '09, 2009, pp. 268–275.

[17] V. K. Garg and A. Agarwal, "Distributed maintenance of a spanning tree using labeled tree encoding." in *Euro-Par*, ser. Lecture Notes in Computer Science, J. C. Cunha and P. D. Medeiros, Eds., vol. 3648. Springer, 2005, pp. 606–616.

[18] M. Dilman and D. Raz, "Efficient reactive monitoring," in *INFOCOM*, 2001, pp. 1012–1019.

[19] S. Meng, L. Liu, and T. Wang, "State monitoring in cloud datacenters," *IEEE Transactions on Knowledge and Data Engineering*, vol. 23, pp. 1328–1344, 2011.

[20] M. Arlitt and T. Jin, "A workload characterization study of the 1998 world cup web site," *Network, IEEE*, vol. 14, no. 3, pp. 30 –37, 2000.

[21] L. A. Adamic and B. A. Huberman, "Zipf's law and the internet," *Glottometrics*, vol. 3, pp. 143–150, 2002.