Accepted for publication in *Proceedings of the 14th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS '12)*
http://www.cs.uwaterloo.ca/conferences/sss2012/

DQMP: A Decentralized Protocol to Enforce Global Quotas in Cloud Environments ⋆

Johannes Behl[1], Tobias Distler[2], and Rüdiger Kapitza[1]

[1]TU Braunschweig [2]Friedrich–Alexander University Erlangen–Nuremberg

Abstract. Platform-as-a-Service (PaaS) clouds free companies of building infrastructures dimensioned for peak service demand and allow them to only pay for the resources they actually use. Being a PaaS cloud customer, on the one hand, offers a company the opportunity to provide applications in a dynamically scalable way. On the other hand, this scalability may lead to financial loss due to costly use of vast amounts of resources caused by program errors, attacks, or careless use.
To limit the effects of involuntary resource usage, we present *DQMP*, a decentralized, fault-tolerant, and scalable quota-enforcement protocol. It allows customers to buy a fixed amount of resources (e. g., CPU cycles) that can be used flexibly within the cloud. DQMP utilizes the concept of diffusion to equally balance unused resource quotas over all processes running applications of the same customer. This enables the enforcement of upper bounds while being highly adaptive to all kinds of resource-demand changes. Our evaluation shows that our protocol outperforms a lease-based centralized implementation in a setting with 1,000 processes.

1 Introduction

Cloud computing is considered a fundamental paradigm shift in the delivery architecture of information services, as it allows to move services, computation, and/or data off site to large utility providers. This offers customers substantial cost reduction, as hard- and software infrastructure needs not to be owned and dimensioned for peak service demand. With Platform-as-a-Service (PaaS) clouds like Windows Azure [1] and Google App Engine [2] providing a scalable computing platform, customers are able to directly deploy their service applications in the cloud. In the ideal case, cloud customers only pay for the resources their applications actually use; that is, "... pricing is based on direct storage use and/or the number of CPU cycles expended. It frees service owners from coarser-grained pricing models based on the commitment of whole servers or storage units." [3]

While it is very inviting to have virtually unlimited scalability and pay for it like electricity and water, this freedom poses a serious risk to cloud customers: the use of vast amounts of resources, caused, for example, by program errors,

⋆ The research leading to these results has received funding from the European Union's Seventh Framework Programme (FP7/2007-2013) under grant agreement n°257243 (TClouds project: http://www.tclouds-project.eu/).

attacks, or careless use, may lead to high financial losses. Imagine an unforeseen input leads to a livelock that consumes massive amounts of CPU cycles. The costs for the resources used unintentionally could be tremendous and may even exceed the estimated profits of running the service.

To address this problem, we propose to employ a quota-enforcement service that allows cloud customers to specify global quotas for the resources (e. g., CPU, memory, network) used by their applications. Such a service can be integrated with the cloud infrastructure in order to ensure that the combined usage of all processes assigned to the same customer does not exceed the upper bound defined for a particular resource.

In domains like grid computing, where application demands are predictable, enforcing global quotas can be done statically during the deployment of an application [4]. However, for user-accessed services in a dedicated utility computing infrastructure [5] like a PaaS cloud, this problem needs to be solved at run time once previously unknown services get dynamically deployed. Further, the quota-enforcement service must not impose any specific usage restrictions: processes must be able to freely allocate resources on demand as long as free quota is available. In this respect, the enforced global quota can be compared to a credit-card limit, which protects the owner from overstepping his financial resources while not making any assumptions on when and how the money is spent. All in all, dealing with a dynamically varying number of processes with unknown resource usage patterns makes quota enforcement a challenging task within clouds.

The straight-forward approach would be to set up a centralized service that manages all quotas of a customer and grants resources to applications on demand. However, as shown in our evaluation, such a service implementation does not scale for applications comprising a large number of processes, which is a common scenario in the context of cloud computing. Moreover, additional mechanisms like, for instance, state-machine replication had to be applied in order to provide a fault-tolerant and highly available solution. Otherwise, the quota-enforcement service would represent a single point of failure.

To avoid the shortcomings of a centralized approach, we devised a decentralized quota-enforcement service including a novel protocol named *Diffusive Quota Management Protocol*, short *DQMP*. DQMP is fault-tolerant and highly scalable by design, two properties that are indispensable for cloud environments. Its basic idea is to use the concept of diffusion to balance information about free quotas across all machines hosting a certain application of a customer. By distributing quota information, the permissions to allocate resources can be granted via local calls. Our service offers a simple and lightweight interface that can be easily integrated to extend existing infrastructures with quota-enforcement support. An evaluation of our prototype with up to 1,000 processes residing on 40 machines shows that DQMP scales well and outperforms a centralized solution.

The remainder of this paper is structured as follows: Section 2 discusses related approaches, Section 3 presents the architectural components of our quota-enforcement service, Section 4 outlines the concept of diffusive quota enforcement and presents the DQMP protocol, Section 5 presents results gained from an experimental evaluation of our prototype, and Section 6 concludes.

2 Related Approaches

Whereas earlier work on diffusion algorithms and distributed averaging addressed various areas such as dynamic load balancing [6,7,8], distributing replicas in unstructured peer-to-peer networks [9], routing in multihop networks [10] and distributed sensor fusion [11], none of them handles quota enforcement. Karmon et al. [12] proposed a quota-enforcement protocol for grid environments that relies on a decentralized mechanism to collect information about free resource quotas as soon as an application issues a demand. In contrast, our protocol proactively balances such information over all machines serving a customer, which allows granting most demands for free quota instantly. Furthermore, this paper goes beyond [12] in extending fault tolerance and in discussing how to integrate with cloud computing. Raghavan et al. [13] proposed an approach targeting distributed rate limiting using a gossip inspired algorithm in cloud-computing environments. They specifically focus on network bandwidth and neglect fault tolerance. Pollack et al. [14] proposed a micro-cash–inspired approach for disk quotas that provides lower overhead and better scalability than centralized quota-tracking services. A quota server acts as a bank that issues resource vouchers to clients. Clients can spend fractions of vouchers to allocate resources on arbitrary nodes of a cluster system. For good resource utilization and to prevent overload of the quota server bank, this requires previous knowledge about the resource demand. Gardfjäll et al. [15] developed the SweGrid accounting system that manages resources via a virtual bank that handles a hierarchical project namespace using branches. Based on an extended name service, each branch can be hosted on a separate node. This approach requires explicit management to be scalable and misses support for fault tolerance. Furthermore, there are distributed lock systems [16,17] that provide fault-tolerant leases based on variants of the Paxos algorithm. Contrary to the presented approach, they are dedicated to manage low volume resources like specific files. As shown by the evaluation, our decentralized protocol scales above such replicated service solutions.

3 Architecture

In this section, we present the key components of our quota-enforcement service which is realized on basis of DQMP and explain how these components interact with existing cloud infrastructures.

3.1 Host Architecture

DQMP uses a decentralized approach to manage the resource quota of customers. It distributes information about free quota units across the machines running applications of the same customer, providing each machine with a *local quota*. Quota enforcement in DQMP spans two levels: (1) At the host level, a *resource controller* guarantees that the local resource usage of an application process does not exceed the local quota. (2) At the global level, a network of *DQMP daemons* enforces a *global quota* by guaranteeing that the sum of all local quotas does not exceed the total quota for a particular resource, as specified by the customer.

Figure 1 shows the basic architecture of a PaaS cloud host that relies on our protocol to enforce quota for two customers A and B. For each of them, a

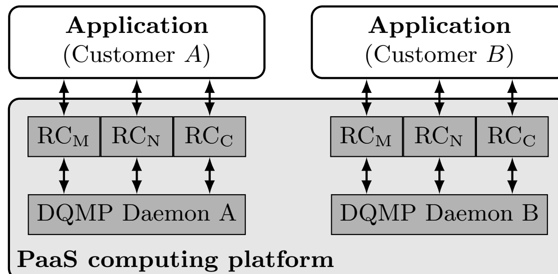


Fig. 1. Basic architecture of a PaaS cloud host running DQMP to enforce resource quotas: quota requests issued by applications of different customers are handled by different DQMP daemons relying on a set of resource controllers (e. g., for memory usage ($\mathrm{RC_M}$), network transfer volume ($\mathrm{RC_N}$), and CPU cycles ($\mathrm{RC_C}$)).

separate DQMP daemon is running on the host. Each DQMP daemon is assigned a set of resource controllers (RC_*) which are responsible for enforcing quotas for different resource types (e. g., memory, network, and CPU).

Resource Controller In general, PaaS computing platforms provide means to monitor the resource consumption of an application process [18]. For DQMP, we extend these mechanisms with a set of resource controllers, one for each resource type. Each time an application seeks to consume additional resources, the corresponding resource controller issues a resource request to its local DQMP daemon and blocks until the daemon grants the demand.

DQMP Daemon A cloud host executes a separate DQMP daemon for every customer executing at least one application process on the host; that is, a DQMP daemon serving a certain customer is only executed on a host when there actually runs a process that may demand resource quota. The main task of a DQMP daemon is to fulfill the resource demands of its associated resource controllers. To do so, the daemon is connected to a set of other DQMP daemons (assigned to the same customer) that run on different cloud hosts, forming a peer-to-peer network. For the remainder of this paper, we will refer to daemons connected in a DQMP network as *nodes*. Moreover, the first node that joins the network is called *quota manager*. It serves as a stable access point for the infrastructure, since the composition of a DQMP network is dynamic as nodes join and leave depending on whether their local machines currently host processes for the customer.

3.2 Node Registry

In addition to the DQMP components running on the same hosts as the customer applications, we provide a *node-registry* service that manages information about all nodes (i. e., DQMP daemons) assigned to the same customer. We assume the node registry to be implemented as a fault-tolerant service; for example, by using multiple registry instances. When a new node joins the DQMP network, the registry sets up an entry for it. As each node periodically sends a heartbeat message the registry is able to garbage collect entries of crashed nodes. When a node leaves the DQMP network (e. g., due to the last local application process having been shut down), the node instructs the registry to remove its entry.

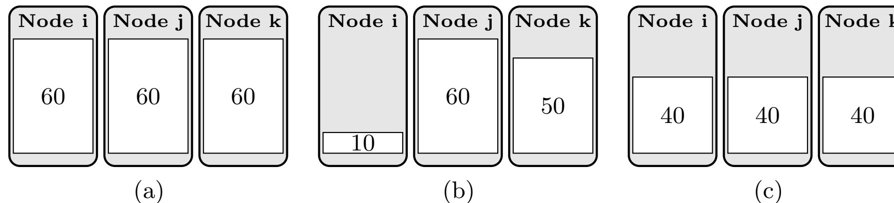


Fig. 2. Example scenario for diffusion-based quota balancing: (a) The local free quotas are balanced across nodes. (b) Processes on nodes i and k demand resources → the diffusion of quota starts. (c) The free quotas have been rebalanced.

4 The DQMP Protocol

This section presents the algorithms used by our decentralized quota-enforcement protocol DQMP to enforce global resource quotas of customers. In addition to a description of the basic protocol, we also discuss extensions for fault tolerance.

4.1 Diffusion-based Quota Balancing

We give a basic example scenario to outline how the general concept of diffusion is applied to balance free global quota information. In this example, three machines have been selected to host the application of a customer. For simplicity, we examine the diffusive balancing process of a single resource quota.

Each node (i. e., DQMP daemon) in the DQMP network is connected to a set of *neighbor nodes* (or just "*neighbors*"). Quota balancing is done by pairwise balancing the free local quota of neighbors. As neighbor sets of different nodes overlap, a complete coverage is achieved. In our example (see Figure 2), nodes i and j form a pair of neighbors, and nodes j and k form another pair of neighbors. At start-up, the global quota of the customer (180 units in our example) is balanced over all participating nodes (see Figure 2a).

When the application starts executing, the resource controller at node i demands 50 resource units and the resource controller at node k demands 10 units. Figure 2b shows that nodes i and k react by reducing the amount of locally available free quota q. Thus, both nodes can grant their local resource demands immediately. Changing the amounts of free quota starts the diffusive quota-balancing process and causes nodes i and k to exchange quota information with other nodes; in this case node j. As the free quota of node j exceeds the free quota of node i (i. e., $q_j > q_i$), $\lceil \frac{q_j - q_i}{2} \rceil$ quota units are migrated to i. The same applies to nodes j and k which, again, leads to different amounts of free quota on nodes i and j. As a result, further balancing processes are triggered and balancing continues until equilibrium is reached. The equilibrium (see Figure 2c) enables node i to be well prepared for future resource demands, as its amount of free quota has risen to the global average of 40.

In case a resource controller issues a resource demand that exceeds q, a node obtains the requested quota by successively reducing q after each balancing process. As soon as the node has collected the full amount, it grants the resource demand to the resource controller.

```
def initial_connect (nodes):
  for node in nodes:
    if node.connect(self ):
      neighbors.append(node)
      if level == None
          or level > node.level:
        level = node.level + 1
        uplink = node

def connect(node):
  if node not in neighbors:
    neighbors.append(node)
    return true
  return false
```

Fig. 3. Connecting nodes

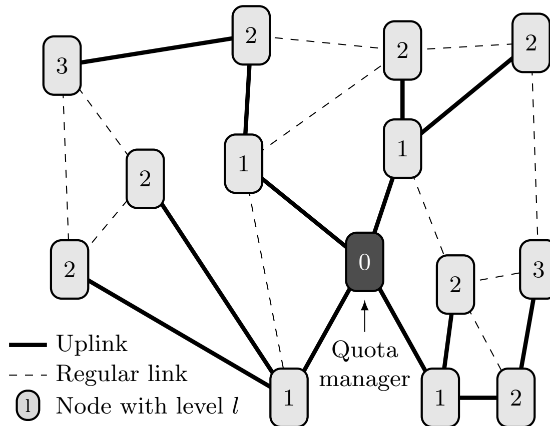


Fig. 4. Example tree in a DQMP network

Using discrete quota, there might be an imbalance of one unit between two neighboring nodes if $mod(\sum q, n) \neq 0$ (n is the total number of nodes), causing balancing to never stop. To avoid this, we restrict balancing to differences above one unit. As a result, this introduces a potential system-wide gradient, which we cope with using probabilistic migration [19]. This strategy migrates small amounts of quota with a certain probability, even if the imbalance is not reduced.

4.2 Basic Protocol

This section describes the basic DQMP protocol. We assume a fail-stop behavior of nodes and the reliable detection of node and connection failures.

Connection Process When a new application is deployed, our quota-enforcement service starts local DQMP daemons on the corresponding hosts and selects one of these nodes to be the quota manager (see Section 3.1). Then, our service supplies all nodes of this first set with the addresses of all other nodes. Next, each node establishes a connection to some of the other nodes, adding them to its neighbor set (see `initial_connect()` in Figure 3 and Table 1).

During this procedure, every node determines its *level* in a tree (see Figure 4) that is formed as a by-product of the connection process. At first, only the quota manager (representing the tree root) is part of the tree and is therefore assigned level zero. Next, all other nodes join the tree using the following algorithm: (1) A node collects the level information of all of its neighbors. (2) It selects the neighbor n that has the lowest level l_n (i. e., the node with the smallest distance

Field	Description
level	Level in the quota tree
neighbors	List of neighbors, where each entry is a triple of **connection**, **counter**, and **level**
quota	Available local resource quota
consumed	Consumed resource quota (see Section 4.3)

Table 1. Data structures managed by a DQMP daemon

```
def do_balancing():
  for n in neighbors:
    free = quota
    # ask other node how to change my quota
    quota += n.balance(free)
```

```
def balance(remote_free):
  free = quota
  avg = (free + remote_free) / 2
  quota += avg − free
  return −(avg − free)
```

Fig. 5. Simplified quota balancing process

to the tree root) to be its parent node in the tree. From now on, we refer to the connection to n as the *uplink*; in Section 4.3, we investigate how the uplink is used to provide fault tolerance. (3) The node sets its own level to $l_n + 1$.

When a node has connected a predefined number of neighbors, it sends an announcement including its contact details and level information to the node registry managing a list of nodes assigned to the customer (see Section 3.2). In case the application of a customer scales up capacity by starting processes on additional hosts, newcomers query the node registry for addresses of nodes in the DQMP network. This information is then used as input for `initial_connect()`.

Quota Balancing When the set of initial nodes is connected, nodes can be provided with quota by simply initializing the quota manager's local free quota with the amount of globally granted quota. In consequence, the diffusion process starts and every node balances its free quota with all connected neighbors.

Figure 5 outlines the basic balancing process, organized in rounds, each comprising a single call to `do_balancing()`. During a round, for each neighbor, a node d determines the amount of free quota and sends it to the neighbor via `balance()`. This method adjusts the free quota at the neighbor and returns the amount by which to change the local free quota of d. The round ends when d has balanced quota with each of its neighbors. Note that quota balancing with a neighbor only takes a single message round-trip time.

If the local free quota has changed during a round of balancing, a node immediately starts another round. Otherwise, the next round is triggered when the node receives a demand from a local resource controller or when the quota exchange with another node modifies the local free quota.

4.3 Extension for Fault Tolerance

In this section, we describe how to extend the basic protocol presented in Section 4.2 in order to tolerate node failures.

General Approach To handle faults, every node maintains a counter for each neighbor link. This link counter represents the net amount of free quota transferred to the neighbor and is updated on each quota exchange via the corresponding link: if a node passes free quota to a neighbor, it increments the local link counter by the amount transferred; the neighbor decrements its counter by the same amount. A negative counter value indicates that a node has received more free quota over that link than the node has passed to the neighbor.

When a node crashes, all connected neighbors detect the crash: each neighbor removes the crashed node from its neighbor set and adds the counter value of the failed link to its local amount of free quota (see Figure 6). This way, the free quota originally held by the crashed node is reconstructed by all neighbors,

```
def fix_crashedNode(neighbor):
    quota += neighbor.counter
    neighbors.remove(neighbor)
    # check if uplink is concerned
    replace_crashedNode()
```

Fig. 6. Recovery after neighbor crash

```
def do_balancing():
 for n in neighbors:
   free = quota
   if n.level < level:
     # pass the consumed quota
     # up to the root
     n.counter += consumed
     result = n.balance(id, free,
                        consumed)[0]
     consumed = 0
   else:
     # receive consumed quota
     # from lower nodes
     (remote_consumed, result) =
                  n.balance(id, free)
     n.counter -= remote_consumed
     consumed += remote_consumed

   n.counter -= result
   quota += result
```

Fig. 7. Issuing a balancing request

```
def balance(id, remote_free,
            remote_consumed = 0):

 neighbor = neighbors[id]
 free = quota
 avg = (free + remote_free) / 2

 # handle the consumed quota
 if neighbor.level < level:
   remote_consumed = consumed
   neighbor.counter += remote_consumed
   consumed = 0
 else:
   neighbor.counter -= remote_consumed
   consumed += remote_consumed

 # balance the remaining quota
 if remote_free < 0 and free < 0:
   # nothing left on both sides
   return (remote_consumed, 0)
 elif remote_free < 0 or free < 0:
   # take care of negative quotas
   # [...]
 else: # free quota on both sides
   quota += avg - free
   neighbor.counter -= avg - free
   return (remote_consumed,
           -(avg - free))
```

Fig. 8. Responding to a balancing request

requiring no further coordination. Note that such a recovery may temporarily leave single nodes with negative local free quota. However, the DQMP network compensates this by quickly balancing quota among remaining nodes.

Consumed Quota So far, this approach is only suitable for refundable quota like disk space, since link counters are unaware that non-refundable quota, like CPU cycles, transferred to a node may have been consumed by a local application process. Thus, neighbors would reassign more free quota than the crashed node actually had. To address this, nodes gather and distribute information about consumed quota, and adjust their link counters to prevent its reassignment.

For each resource, a node maintains a `consumed` counter (see Table 1) that is updated whenever a local application process consumes quota. Each node periodically reports the value of its `consumed` counter to its uplink, which in turn passes it to its own uplink, and so on, all up to the quota manager. Having reported the consumed quota, a node increments its uplink link counter by the amount announced; the uplink in turn decrements its link counter by the same value, similar to the modifications triggered during quota balancing. As a result, link counters are adjusted to reflect the reduced global free quota. Figures 7 and 8 show updated listings of the balancing process presented in Figure 5.

Handling Cluster Node Failures Link counters are an easy and lightweight mean to compensate link crashes and single node failures. They also allow tolerating multiple crashes of directly connected nodes, because adjacent nodes can be seen as one large node with many neighbors. In case a node set is separated from the rest of the network, the node set that is not part of the quota-manager partition eventually runs out of quota, since free quota is always restored in the direction of its origin (i. e., the quota manager). However, after reconnection, the balancing process re-distributes the free quota, enabling the application processes on all nodes to make progress again. To avoid permanent partitions within the network, the protocol makes use of the level information. When a node except the quota manager and its direct neighbors loses the connection to its uplink, it has to select a node with a lower level than its own as new uplink. Preferably, the node uses one of its current neighbors for that purpose; however, it can also query the node registry (see Section 3.2) for possible candidates. If a suitable uplink cannot be found, the node is shut down properly.

Handling Crashes of the Quota Manager If the quota manager crashes, its neighbors do not consolidate their link counters. If they did, all global quota of a customer would vanish as it has been originally injected via the quota manager. Instead, all links to the quota manager are marked *initial links* and are therefore ignored during failure handling, allowing the network to proceed execution.

However, we assume a timely recovery of the quota manager as an application cannot be provided with additional quota while this node is down. We therefore assume that its state can be restored (e. g., using a snapshot). Note that the state of the quota manager to be saved is small: it only includes the set of neighbor addresses as well as the `quota`, `consumed`, and `counter` values (see Table 1) for every managed resource, making frequent snapshots and a fast recovery feasible.

At restart, the quota manager reconnects all level-one nodes. In case of one or more of them having crashed in the meantime, it starts the regular failure handling. At this point, we cannot tolerate network partitions between the quota manager and its neighbors, as this would lead to a duplication of free quota.

5 Evaluation

We evaluate DQMP on basis of a prototype implemented in Java. The tests are performed on 40 hosts, all equipped with 2.4 GHz quad-core CPU, 8 GB RAM, and connected over switched Gigabit Ethernet. Each host executes up to three Java virtual machines (JVMs) to support the simulation of larger networks. In this set-up, raw ping times range from 0.2 to 0.5 ms and simple Java RMI method calls take between 0.7 and 1.0 ms. On top of the physical network, two DQMP networks, consisting of 100 and 1,000 nodes, are simulated, with the maximum number of neighbors set to 6. Comparison measurements show, that simulating up to nine nodes within a single JVM has no significant impact on the results.

Test runs are performed as follows: After the DQMP network is built up, a quota amount of 50,000 units per node is injected. When the initial equilibrium is established, all nodes are instructed to begin with the execution of the actual test. After a test has finished, the local results of the nodes are collected. Except time charts, all presented results are the average of at least three test runs.

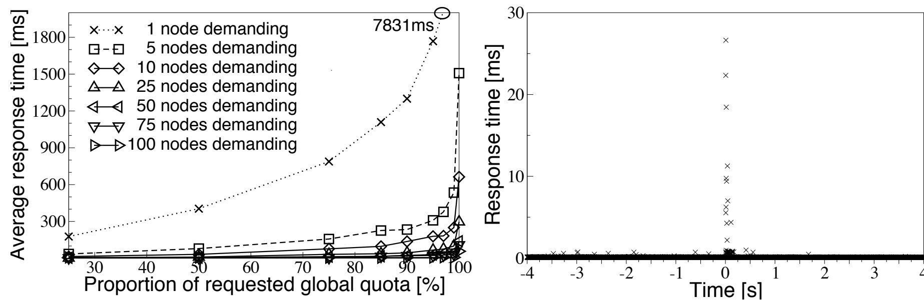


Fig. 9. Response times for single demands of varying amounts from varying parts out of 100 nodes

Fig. 10. Response times of a single test run with 100 constantly requesting nodes and a crash of 25 nodes at $t = 0$

5.1 Response Time Behavior of DQMP

Single Demands In the first test, we examine the response times of DQMP for single demands within the small network containing 100 nodes. In this scenario, a subset of nodes orders a predefined amount of quota at the same time. The proportion of demanding nodes is raised stepwise from 1% to 100% and the overall amount of quota requested by this proportion is varied between 25% and 100%. This means that in one case, for instance, a single node requests the entire quota available and in another case, each of 100 nodes requests 1% of it.

From the results, as depicted in Figure 9, it can be inferred that the decisive factor for the performance of our protocol is the ratio between the free local quota held by each node and the size of the local demand: the smaller the demand compared to the local quota, the faster it can be satisfied. Since DQMP aims to an even distribution of free quota over all nodes, the demand size can be put into relation to the globally free quota: if demands of single nodes exceed the average size of free quota held by each node to a great extent, it is likely that quota has to be transferred not only from nearer nodes but also from farther ones to satisfy the demand. For instance, if a single node asks for the entire available quota, every quota unit in the network has to reach the same destination. With our settings, this takes about 7.8 seconds and 770 balancing rounds per node. However, this case is not realistic as only such nodes participate in DQMP networks that are actually used by processes demanding quota. If 50 nodes request 95% of the overall quota, the provisioning time already drops below 30 ms. Here, it takes about 45 balancing rounds per node until the request is fulfilled and until the network comes to a rest, that is, until no messages are transmitted anymore. Moreover, if only a small amount of the overall quota is needed or a large demand is split between many nodes, DQMP can provide extremely low response times. When a demand of a node can be fulfilled by its local quota, the DQMP daemon is even able to instantly grant the demanded amount, turning the assignment of global quota within a distributed system into a local operation.

Crashes of Nodes After this first evaluation, we now examine how our protocol behaves in the presence of node crashes, since fault tolerance was a primary objective for the design of DQMP. As basis for this evaluation, we choose a

scenario in which nodes demand and release quota constantly. In detail, each node performs the following in a loop: It adds a randomly chosen delta d, with $-10,000 \leq d \leq +10,000$, to its previous quota demand. It ensures that the new demand does not exceed the upper bound b of $50,000$ units, which limits the demand of all nodes combined to 100% of the overall quota injected into the system. According to the calculated value, the node issues a request either demanding new or releasing already granted quota. Subsequently, it waits until the request is fulfilled. Then it sleeps for a randomly chosen time between 25 and 75 ms to simulate fluctuating resource requirements.

Figure 10 shows the course of response times from a single test run with 100 requesting nodes, issuing a total of approximately 14,000 requests within 8 seconds, and an induced crash of 25 nodes at $t = 0$. The first outcome of this test is, that under the given scenario, which simulates the distribution of a large demand over all available nodes, almost all quota requests can be fulfilled locally, leading to a standard response time below 0.2 ms. For the same reason, the processing of most requests is hardly affected by crashes of neighbors. Quota releases are inherently not affected at all anyway. Consequently, despite the crash of 25% of the nodes, there are only 4 requests for which it took between 10 and 30 ms to process them and 8 requests that lie in the range between 1 and 10 ms. Thus, the balancing process of DQMP is able to compensate node crashes very quickly by redistributing the quota over all remaining nodes.

5.2 Comparison of Different Architectures

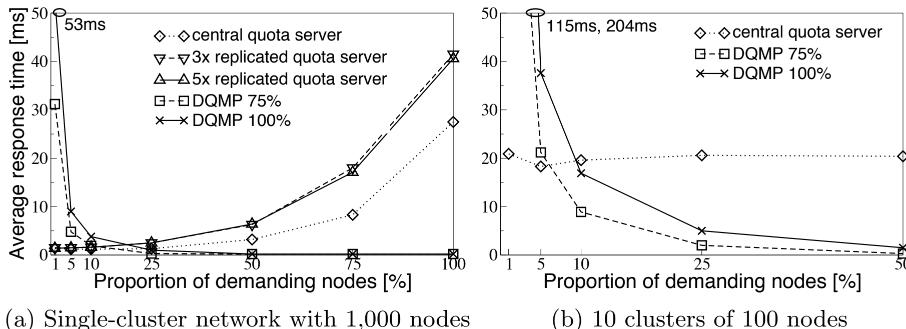
Next, we compare DQMP to other architectures addressing quota enforcement in distributed systems. For this purpose, we implemented a RMI-based quota server and a passively replicated variant of it by means of the group communication framework JGroups[1]. During test runs, the quota server as well as each replica is executed by a dedicated machine. In the following, the term "node" is not confined to DQMP daemons; it also denotes clients in the other architectures.[2]

As scenario for the comparison serves an extended variant of the scenario used for examining the behavior of DQMP in the presence of nodes crashes (see Section 5.1). Different to the previous scenario, here, a network of 1,000 nodes is used and the proportion p of requesting nodes is varied between 1% and 100%. Further, the combined demand of all requesting nodes is limited to 75% of the overall injected quota in one case and to 100% in another. This is achieved by setting the maximum demand of a single node b to $b_{75\%} = \frac{37.500}{p}$ and $b_{100\%} = \frac{50.000}{p}$, respectively. The delta d for every simulated demand change is randomly chosen between $-0.2b$ and $+0.2b$ quota units.

Single-cluster Network For a first comparison, all network connections have similar latencies, just as in the previous tests and just as found within a local

[1] http://www.jgroups.org/

[2] We also implemented a quota-enforcement service based on the coordination service Apache ZooKeeper (http://zookeeper.apache.org/). However, the optimistic lock approach of ZooKeeper is not suitable for the high number of concurrent writes needed in such systems, resulting in some orders of magnitude higher response times.



(a) Single-cluster network with 1,000 nodes (b) 10 clusters of 100 nodes

Fig. 11. DQMP compared to other architectures regarding response times

area network, for instance within a single data center of a cloud provider. The results of this scenario are depicted in Figure 11a. Since response times of the central quota server and its replicated variant are only dependent on the number of quota requests that have to be processed, and particularly are independent of quota amounts, only a single set of results is reported for these architectures.

This test reveals the deficiencies of not completely decentralized systems in terms of scalability: Due to limited resources such as CPU power, memory and bandwidth and due to the contention arising from the shared usage of such resources, these systems have a limited rate they can process requests at. In our settings, for instance, all quota-server–based systems are able to process the requests of a smaller number of requesting nodes within less than 2 ms on average. However, in the presence of 1,000 requesting nodes, a single quota server already requires about 28 ms. Using a more reliable replicated server system makes this even worse. The increased communication overhead leads to an average response time of over 40 ms.

In contrary, using DQMP response times decrease when demands are split up between more nodes. DQMP is able to fulfill requests within an average of 1 ms, and is thus faster than the server systems when the proportion of requesting nodes exceeds 25%. Beyond 50% the response time drops constantly below 0.2 ms. Since the total demand was fixed to either 75% or 100% of the globally injected quota, single demands get smaller with an increasing number of requesting nodes, leading to a higher chance that requests can be fulfilled through the local quotas of the nodes. That is the reason why, as shown by our results, DQMP is even able to outperform a non-saturated central quota server in terms of average response times when demands are distributed over multiple nodes.

Clustered Network Normally, cloud providers do not maintain only a single data center but multiple ones, spread all over the world. These data centers form a clustered network, a network in which groups of well-connected nodes can only communicate among each other over relatively slow connections. To simulate such an environment, respectively wide area networks in general, we assign each out of 1,000 nodes to one of 10 clusters and artificially delay message exchange between nodes from different clusters by 20 ms.

The results, as presented in Figure 11b, suggest the conclusion that a central quota server is not well suited for the scenario described here. The server is located in one of the 10 clusters, which entails that 90% of all nodes experience prolonged delays while communicating with it. Thus, in 90% of all quota requests, demands or releases, the delay of 20 ms is fully added as an offset to the processing time. In case of DQMP, nodes can exchange quota with all of their neighbors in parallel, mitigating the effects of slower connections. Furthermore, all requests that can be fulfilled locally, including all releases, are not affected at all by communication delays. These are the reasons, why DQMP is able to provide better response times than a quota server in this scenario already when only 10% of the nodes demand and release quota.

Protocol Overhead Concerning the protocol overhead of DQMP regarding network transfers, it can be observed that DQMP has completely different characteristics than a traditional quota server. If a quota server is used, each quota request leads to the exchange of two messages, a request message and its reply. In our implementation, the two messages require about 100 bytes. With DQMP instead, requests have only an indirect influence on the balancing process and hence, on the number of messages transferred. For the unrealistic case (see above) that relatively large demands are infrequently issued by a single node, causing, in the worst case, continuous balancing processes all over the network, the ratio between number of requests and messages transferred is unfavorable. With an increasing number of requests, however, the ratio gets more appropriate. In the scenario of 1,000 constantly requesting nodes our protocol requires about 3 kilobytes per request in average. Although this is still more than needed by the quota-server system, it has to be noted, that DQMP provides fault-tolerant operation while a central quota server does not and that network traffic between hosts of the same data center is usually not billed by cloud providers, hence, using DQMP would not generate additional transfer costs for cloud customers.

6 Conclusion

In this paper, we presented DQMP, a decentralized quota-enforcement protocol that provides the fault tolerance and scalability required by cloud-computing environments. DQMP can help customers of platform services, to prevent themselves from financial losses due to errors, attacks, or careless use causing involuntary resource usage. The utilized diffusion-based balancing of free quota enables customers to enforce global limits on resource usage while retaining flexibility and adaptability regarding the actual local demands within their deployments. Nonetheless, DQMP is not confined to this application. Cloud providers can employ it, for example, to restrict customers of their platform or infrastructure services on a global level by enforcing quota for virtual machines. As the evaluation of our prototype implementation shows, DQMP is able to provide better response times than a centralized service in a setting with 1,000 nodes. Moreover, our protocol is well suited for clustered networks as formed by interconnected data centers. Both is important since traditional, not fully decentralized solutions might soon reach their limit as distributed systems get larger and larger.

References

1. Windows Azure Platform. http://www.microsoft.com/windowsazure/
2. Google App Engine. http://code.google.com/appengine/
3. Creeger, M.: Cloud computing: An overview. ACM Queue **7**(5) (2009)
4. Schopf, J.M.: Ten actions when Grid scheduling: the user as a Grid scheduler. In: Grid Resource Management: State of the Art and Future Trends. Kluwer Academic Publishers (2004) 15–23
5. Rolia, J., Cherkasova, L., Arlitt, M., Machiraju, V.: Supporting application quality of service in shared resource pools. Communications of the ACM **49**(3) (2006) 55–60
6. Cybenko, G.: Dynamic load balancing for distributed memory multiprocessors. Journal of Parallel Distributed Computing **7**(2) (1989) 279–301
7. Boillat, J.E.: Load balancing and Poisson equation in a graph. Concurrency: Practice and Experience **2**(4) (1990) 289–313
8. Corradi, A., Leonardi, L., Zambonelli, F.: Diffusive load-balancing policies for dynamic applications. IEEE Concurrency **7**(1) (1999) 22–31
9. Uchida, M., Ohnishi, K., Ichikawa, K.: Dynamic storage load balancing with analogy to thermal diffusion for P2P file sharing. In: Proc. of the 2006 Work. on Interdisciplinary Systems Approach in Performance Evaluation and Design of Computer & Communications Systems. (2006)
10. Tassiulas, L., Ephremides, A.: Stability properties of constrained queueing systems and scheduling policies for maximum throughput in multihop radio networks. In: Proc. of the 29th IEEE Conf. on Decision and Control. (1990) 2130–2132
11. Xiao, L., Boyd, S., Lall, S.: A scheme for robust distributed sensor fusion based on average consensus. In: Proc. of the 4th Intl. Symp. on Information Processing in Sensor Networks. (2005) 63–70
12. Karmon, K., Liss, L., Schuster, A.: GWiQ-P: An efficient decentralized grid-wide quota enforcement protocol. SIGOPS OSR **42**(1) (2008) 111–118
13. Raghavan, B., Vishwanath, K., Ramabhadran, S., Yocum, K., Snoeren, A.C.: Cloud control with distributed rate limiting. In: Proc. of the 2007 Conf. on Applications, Technologies, Architectures, and Protocols for Computer Communications. (2007) 337–348
14. Pollack, K.T., Long, D.D.E., Golding, R.A., Becker-Szendy, R.A., Reed, B.: Quota enforcement for high-performance distributed storage systems. In: Proc. of the 24th Conf. on Mass Storage Systems and Technologies. (2007) 72–86
15. Gardfjäll, P., Elmrothaell, E., Elmroth, E., Johnsson, L., Mulmo, O., Sandhol, T.: Scalable grid-wide capacity allocation with the SweGrid Accounting System (SGAS). Concurrency and Computation: Practice and Experience **20**(18) (2008) 2089–2122
16. Hupfeld, F., Kolbeck, B., Stender, J., Högqvist, M., Cortes, T., Marti, J., Malo, J.: FaTLease: scalable fault-tolerant lease negotiation with Paxos. In: Proc. of the 17th Intl. Symp. on High Performance Distributed Computing. (2008) 1–10
17. Burrows, M.: The Chubby lock service for loosely-coupled distributed systems. In: Proc. of the 7th Symp. on Operating Systems Design and Implementation. (2006) 335–350
18. Weissman, C.D., Bobrowski, S.: The design of the Force.com multitenant Internet application development platform. In: Proc. of the 35th SIGMOD Intl. Conf. on Management of Data. (2009) 889–896
19. Douglas, S., Harwood, A.: Diffusive load balancing of loosely-synchronous parallel programs over peer-to-peer networks. ArXiv Computer Science e-prints (2004)