QoS-enabled Distributed Mutual Exclusion in Public Clouds

James Edmondson[1], Doug Schmidt[1], and Aniruddha Gokhale[1]

Dept. of EECS, Vanderbilt University, Nashville, TN 37212, USA
Email: {james.r.edmondson,doug.schmidt,a.gokhale}@vanderbilt.edu

Abstract. Popular public cloud infrastructures tend to feature centralized, mutual exclusion models for distributed resources, such as file systems. The result of using such centralized solutions in the Google File System (GFS), for instance, reduces scalability, increases latency, creates a single point of failure, and tightly couples applications with the underlying services. In addition to these quality-of-service (QoS) and design problems, the GFS methodology does not support generic priority preference or pay-differentiated services for cloud applications, which public cloud providers may require under heavy loads.
This paper presents a distributed mutual exclusion algorithm called Prioritizable Adaptive Distributed Mutual Exclusion (PADME) that we designed to meet the need for differentiated services between applications for file systems and other shared resources in a public cloud. We analyze the fault tolerance and performance of PADME and show how it helps cloud infrastructure providers expose differentiated, reliable services that scale. Results of experiments with a prototype of PADME indicate that it supports service differentiation by providing priority preference to cloud applications, while also ensuring high throughput.
keywords: mutual exclusion, public cloud, QoS, file systems

1 Introduction

The Google File System (GFS) was designed to support the sustained file throughput capacities of the Google search engine [1–3]. GFS provides high throughput in a single cluster of thousands of computers, each servicing the Google search engine. Although the GFS scaled well to hundreds of terabytes and a few million files in append-mode (GFS does not support overwriting a file), other quality-of-service (QoS) properties (*e.g.*, latency, throughput of small files—which is common in many applications, and differentation amongst applications) were not the focus of its initial design.

Scalability problems with GFS began appearing when the centralized master server was forced to process tens of petabytes worth of data requests and appends [2]. As a short-term solution, Google engineers used a centralized master server to manage a *cell* of the overall cluster. Although this approach provided some fault tolerance against the single master failing, some failures still occurred, and throughput and scalability suffered [1].

As Google grew, so did its list of services and applications. Since GFS focused on throughput rather than latency and scalability, performance issues appeared with certain applications, such as GMail, Youtube, and Hadoop [1]. Google's temporary solution to overcome this problem was the BigTable application, which was layered atop GFS and packed small files into the large 64 MB file metadata that had been in place since their Internet crawler was first deployed [1, 2].

For cloud applications (such as Youtube) that can be buffered, the latency of the GFS system has mostly been acceptable. For applications with file accesses and writes on the cloud, however, Google is looking into replacements for GFS that provide better QoS [1]. Deferring these changes can be costly for applications that have a GFS-like interface or these applications face mounting integration issues.

Figure 1 shows the scalability problems of a centralized server for reading and writing to a segregated file system. In this figure, the light-shaded lines rep-

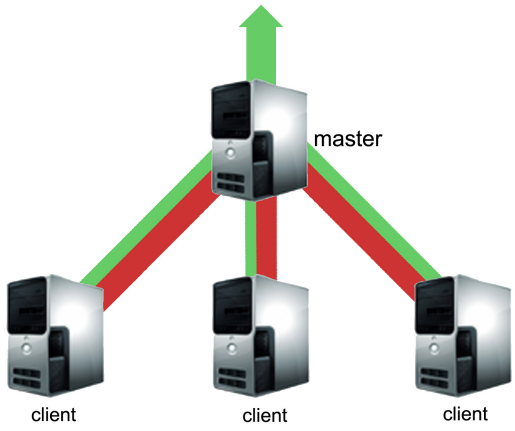

Fig. 1. GFS Centralized Master Controller

resent the computational and bandwidth resources that are utilized and dark represents the wasted resources. According to Google network engineers, the major bottlenecks of the GFS system include the inability to provide native overwrite operations (GFS supports only append mode and applications have to work around this), the 64 MB metadata for even small files, and the centralized master controller that every request and write goes through in each cluster cell. The centralized master controller provides mutual exclusion for read-write operations, garbage collection, and replication and persistence. Even with extra computing resources, this controller reduces scalability and throughput, and significantly increases latency due to queuing [1].

In addition to the latency bottleneck, reduced overall throughput, and lack of fault tolerance, GFS's centralized architecture also treats all application requests equally. Applying this system to cloud providers with scarce resources and steady client application deployments means there is no built-in differentiation between client priorities according to their payscale or other factors.

In general, a cloud file system should address the following challenges:

1. **Avoid centralized bottlenecks**, such as a master controller which restricts file throughput, and increases latency due to queuing and limited bandwidth through a single host.
2. **Handle failures of nodes and applications**, *e.g.*, GFS requires that master controllers be manually reset by an engineer when they fail [1].
3. **Support aggregated priorities**, *e.g.*, based on expected cost, payment, etc. of cloud applications.

This paper presents a distributed mutual exclusion algorithm called *Prioritizable Adaptive Distributed Mutual Exclusion* (PADME) that can be used to address these challenges in cloud file systems by decentralizing the mutual exclusion problem, as shown in Figure 2. As with Figure 1, the decentralized PADME

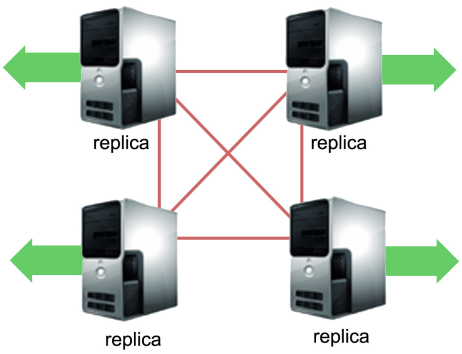

Fig. 2. Scalability of a Decentralized Methodology

algorithm still has a certain amount of overhead (shown in dark) due to the replicas and clients working together and coming to a consensus. In contrast, however, the overall capacity of the decentralized PADME approach scales with the number of participating nodes, rather than being limited by a single master controller.

The remainder of this paper is organized as follows: Section 2 describes the design and capabilities of of PADME algorithm; Section 3 evaluates results from experiments conducted on a simulated message-oriented prototype of the PADME algorithm over shared memory; Section 4 compares our work on PADME with related research; and Section 5 presents concluding remarks and lessons learned.

2 Distributed Mutual Exclusion in Public Clouds

This section presents an algorithm called *Prioritizable Adaptive Distributed Mutual Exclusion* (PADME) that we developed to meet the cloud file system challenges described in Section 1. The PADME algorithm performs two main operations:

1. It maintains a spanning tree of the participants (*e.g.*, applications, customers, or anything that wants access to the file system) in the network. The spanning tree is based upon the customer or application priorities and the root of the spanning tree will be the highest priority entity in the system. Depending on the network topology and customer demands this root can and will change during runtime operations.
2. It enforces mutual exclusion according to user-specified models and preferences for messaging behavior. The models supported by the PADME algorithm include priority differentiation and special privileges for intermediate nodes in the spanning tree (intermediate nodes are nodes between requesting nodes). Each model may be changed during runtime if required by the cloud provider, applications, or users.

Below we describe the PADME algorithm and show how it can be implemented efficiently in cloud middleware platforms or cloud applications, wherever the distributed mutual exclusion is appropriate.

2.1 Building the Logical Spanning Tree

PADME builds a logical spanning tree by informing a cloud participant (*e.g.*, an application that is using the cloud infrastructure to manage the file system) of its parent. Under ideal circumstances, all the spanning tree construction should be performed in the cloud infrastructure without affecting user applications.

A participant need not be informed of its children as they will eventually try to contact their parent, establishing connections on-demand. PADME uses this same mechanism to reorder the tree when optimizing certain high priority participants. Each participant is responsible for reconnecting to its parent through the cloud API, middleware, or however the cloud offers file system services.

To establish which parent to connect to, PADME's joining process multicasts its priority and waits for responses during a (configurable) timeout period. The closest priority above the joining process becomes the parent. The process can arbitrarily connect to a different parent later, in which case the parent will need to remove pending requests from the child. This step can be postponed until the file access permission returns to this node for the specific requester to eliminate the need for a parent to care about children moving to other parents. If the connection no longer exists the token is sent back up to higher priority processes in the logical spanning tree of cloud participants.

Cloud middleware developers could decide to let each cluster form its own priority-based spanning tree and connect roots of each tree to form the cloud file

system mutual exclusion tree. The PADME algorithm presented in Section 2.2 will work with any spanning tree as long as connected nodes can communicate. Section 2.3 covers PADME's fault tolerance support. Figure 3 shows the construction of such a spanning tree out of priority information. In this case, the

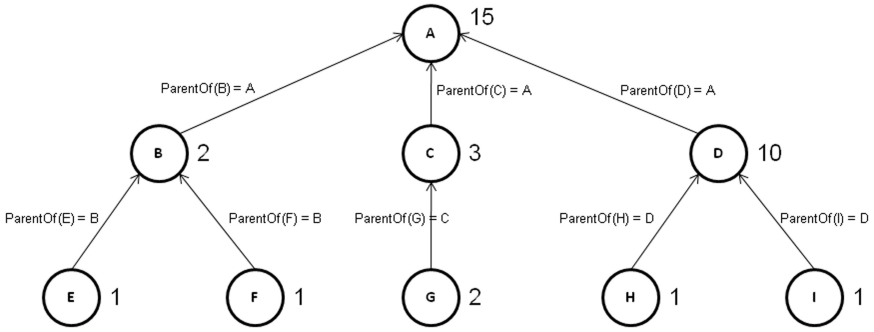

Fig. 3. Building a Balanced Spanning Tree

tree is balanced, though it need not be. During runtime, an application or user may add or remove a participant, rename participants, or conduct other such operations to organize the intended tree and dynamically respond to changes in request load or priority changes.

PADME's tree building phase requires updating affected participants with parent information (*i.e.*, informing them of the direction of the logical root of the tree). The messaging overhead of maintaining a logical spanning tree is not included in our algorithm details because cloud providers can simply build a spanning tree once manually if desired. For example, the cloud provider may know that the application deployments are static, or the provider is anticipating a specific scenario like spikes of file access during a major sporting event and wants to give preference to this activity during the championship match.

Even in statically assigned spanning trees, higher priority cloud participants should be pushed towards the root to give them preferential service. The logical token will be passed back to the root of the tree before continuing on to the next participant in our algorithm. Participants at higher levels of the tree experience lower message complexity, faster synchronization delay, better throughput, and even higher QoS for the target system and a preference for high priority customers, as shown in Section 2.3.

2.2 Models and Algorithm for Distributed Mutual Exclusion

Before accounting for faults, PADME requires just three types of messages: *Request, Reply,* and *Release*. A Request message is made by a participant that

wants to acquire a shared resource, such as cloud file system access. A Request message traverses up the spanning tree from the participant node to the root via its parent and ancestor nodes. A Reply message is generated by the root after access to the resource is granted. The Reply message traverses from the root to the requesting participant node. The Release message traverses up the tree from the node that holds the shared resource towards the root once the node is ready to release the resource. The interaction of these messages is shown in Figure 4.

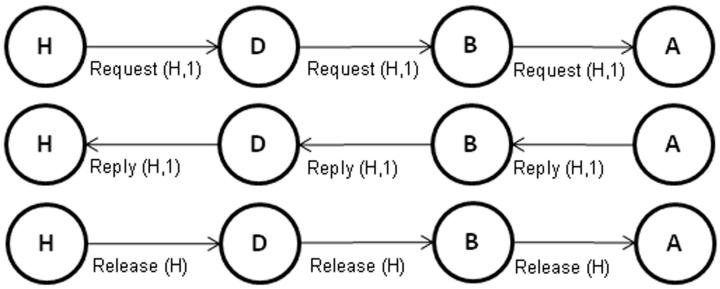

Fig. 4. PADME Messages for Mutual Exclusion

PADME supports four models (*Priority Model, Request Model, Reply Model,* and *Release model*) that describe the semantics of actions performed by any participant in the spanning tree that receives one of the three types of messages, as well as QoS differentiation that must be supported. The latter three models are named according to whether an intermediate participant can enter its own critical section, *i.e.*, exclusive access to the cloud's distributed file system resource, upon receipt of the message type. These models stem from our approach to distributed mutual exclusion and optimizations that allow shorter synchronization delay between critical section entries and improved QoS via user-specified requirements to middleware.

The configurations of the Request, Reply, and Release models may be changed at runtime to yield different QoS, including

- **Higher critical section throughput** – *i.e.*, the number of file accesses possible to the cloud's distributed file system,
- **Changes in fairness** – *e.g.*, going from preferring higher priority participants to giving everyone a chance at the critical section – a requirement of our motivating scenario in Section 1,
- **Fewer priority inversions** – *i.e.*, alleviating the situation where a low priority participant gets a critical section entry before a high priority participant, even though a critical section request from a higher priority participant exists, and

– **Lower average message complexity** – *i.e.*, fewer messages being required per critical section entry.

These four models are described below and each are integral components in the algorithm that may be tweaked to affect performance, usually at the cost of possible priority inversions.

Request Models PADME provides two Request Models: *Forward* and *Replace*. The Forward Request Model requires a parent to immediately forward all requests to its own parent. The Replace Request Model requires a parent to maintain a priority queue of child requests, which should have the same Priority Model as the root participant. Under the Replace Request Model, a node only sends a Request to its parent if there are no Request messages in its priority queue, or if the new Request is of higher priority than the last one that was sent. The Replace Request Model is slightly harder to implement, but it results in messages only being sent when appropriate and may alleviate strain on the root node. It also will result in less message resends if a parent node fails.

Reply Models PADME provides two Reply Models: *Forward* and *Use*. The Forward Reply Model requires a parent to immediately forward a reply to its child without entering its own critical section, regardless of whether or not it has a request pending. The Use Reply Model allows a parent P_c to enter its critical section upon receiving a Reply message from its parent P_p, if P_c currently has a Request message outstanding.

Release Models PADME provides two Release Models: *Forward* and *Use*. The Forward Release Model requires a participant to immediately forward a Release message to its parent without entering its own critical section, regardless of whether it has a request pending. The Use Release Model allows a participant to enter its critical section when it receives a Release message from one of its children, if the participant has an outstanding Request pending.

When applying the Use model, the participant must append its identifier onto the Release message if it entered its critical section (as shown in Figure 5, where each participant appends their release information to their parents when

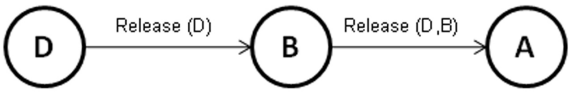

Fig. 5. Appending Identifier to Release Message

the critical sections have already been entered), which may result in a Release message containing multiple instances of the participant identifier. Consequently, appropriate data structures should be used to allow for these duplicates (*e.g.*, multisets). These duplicates enable proper bookkeeping along the token path since up to two Request messages may require removal from each affected priority queue.

Priority Models PADME provides two Priority Models: *Level* and *Fair*. The Level Priority Model means that one Request of the tuple form Request $< I_m, P_m, C_m >$ should be serviced before Request $< I_n, P_n, C_n >$ if $P_m < P_n$. P_x stands for the priority of the participant identified by I_x, and C_x refers to the request id or clock. If a tie occurs, the clocks C_x are compared first and then the identifiers. This ordering does not guarantee the absence of priority inversions, and priority inversions may happen when the token is in play (walking up or down the tree).

The Fair Priority Model means that one Request of the form Request $< I_m, P_m, C_m >$ should be serviced before Request $< I_n, P_n, C_n >$ if $C_m < C_n$. Upon a tie, the priority levels are compared, followed by the identifiers. The Fair Priority Model will result in all participants eventually being allowed into a critical section (assuming bounded critical section time and finite time message delivery), whereas the Level Priority Model makes no such guarantees.

Overview of PADME's Mutual Exclusion Algorithm When a participant needs to enter its critical section (*e.g.* an agent is requesting exclusive access for writing to a cloud file system), it sends a Request message to its parent, who then forwards this Request up to its parent, until eventually reaching the root node. The Request message is a tuple of the form Request $< I, P, C, D >$, where I is the identifier of the requesting participant, P is the priority level (level), C is a timer or request id, and D is a user data structure that indicates the shared resource id (*e.g.*, the name of the file that will be accessed in the cloud file system) and any other data relevant to business logic. There is no reason that any of these variables be limited only to integers. For more information on the election of cloud entity and volume identifiers that may be useful for a cloud file system implementation, please see the Renaming Problem [4].

The choice of a timer mechanism (also known as a request id) may result in varying ramifications on the Fair Priority Model, discussed in Section 2.3. A timer should be updated (1) only when sending a Request or (2) any time a Request, Reply, or Release message with the highest time that of the agent who is receiving message or the time indicated in the message sent. The latter method will result in time synchronization across agents which can be helpful in synchronizing fairness in late joining agents or when switching from Level Priority Model to Fair Priority Model. Resending a Request does not increase the local request count. A Request may be resent if the parent participant faults or dies to ensure that a Request is serviced eventually by the root.

The root participant decides which Request to service according to a priority mechanism. After determining who gets to enter their critical section next, a Reply message is sent of the form Reply $< I, C >$ or $< I, C, D >$ where I is once again the identifier of the requesting participant, C is the count of the Request, and D is an optional parameter that may indicate business logic information, *e.g.*, the name of the file to be overwritten. Once a Reply message reaches the intended requesting participant, the requesting participant enters its critical section.

Upon exiting the critical section, the requesting participant must send a Release message to its parent participant, who forwards this Release message to its parent until the root receives the message. Release messages have the form Release $< I_0, I_1, \ldots I_n >$ or $< I_0, D_0, I_1, D_1, \ldots I_n, D_n >$ where $I_0, I_1, \ldots I_n$ is a list of participant identifiers that used their critical section along this token path, and $D0, D1, \ldots Dn$ is a parameter that may indicate business logic information *e.g.*, the frequency that is being released. The root participant and any participant along the token path should remove the first entry of each identifier in$< I_0, I_1, \ldots I_n >$ before forwarding the Release to its parent for proper bookkeeping.

2.3 QoS Properties of the PADME Algorithm

PADME's Request, Reply, Release, and Priority Models described in Section 2.2 are orthogonal and may be interchanged by the user to accomplish different QoS, higher fault tolerance, reduced message complexity at key contention points, or critical section throughput during runtime. Each combination has certain QoS properties that may fit an application's needs better than the others, *e.g.*, each has certain synchronization delay characteristics, throughput, and even message complexity differences during fault tolerance. To simplify understanding of the different combinations of these models, we created a system of model combinations that we call Request-Grant-Release settings that codify these combinations.

Non-fault Tolerance Case PADME's most robust Request-Reply-Release setting is the Replace-Use-Use model, which corresponds to the Replace Request Model, Use Reply Model, and Use Release Model. The Replace-Use-Use setting requires each participant to keep a priority queue for child Requests (described further in Section 2.2), but its primary purpose is to limit the number of message resends during participant failures or general faults to only the most important Requests in the queue. Replace-Use-Use is consequently very useful when reducing the number of messages in the network is a high priority.

PADME's Use Reply Model of the Replace-Use-Use combination allows a participant to enter its critical section before forwarding on a Reply message to an appropriate child. The Use Release Model allows a similar mechanism in the opposite direction, on the way back to root. Both of these use models work well in conjunction with the Fair Priority Model to not only decrease synchronization delay (and thus increase critical section throughput) but also favor higher priority participants, as those higher priority participants will be

closer to root and may have up to two chances of entering a critical section along a token path from root to a requestor and back to root.

Even when the Forward-Forward-Forward combination is used, the higher priority participants closer to root will still have lower message complexity and lower average synchronization delay than lower priority participants (*e.g.*, leaf nodes). This results from the token path being longer from the leaf nodes to root. Consequently, placing frequently requesting participants closer to the root node in the logical routing network can result in increased performance (Section 2.3 analyzes message complexity and synchronization delay).

All of PADME's Fair Priority Model-based settings inherently may lead to priority inversions. PADME's Level Priority Model by itself, however, does not eliminate priority inversions. To eliminate priority inversions from occuring in PADME, the Level Priority Model must be used in combination with *-Forward-Forward.

If the settings contain Use Models for either the Reply or Release Models when the virtual token progresses towards an entity, it is possible that a higher priority request may be delayed at the root node that arrived after permission was granted to a lower priority entity. In practice, completely eliminating priority inversions is typically not as important as throughput. Settings that enable the Use Model for both Reply and Release models therefore have much higher throughput proportional to the depth of the spanning tree.

Fault Tolerance Case There are several options for fault tolerance that can be supported by PADME, but we focus on the most straightforward to implement and still be robust to failures. We use a classic approach called a Byzantine view change [5] whenever a root node faults (*i.e.*, becomes unavailable). A Byzantine view change is a type of consensus that requires a majority agreement for electing a primary node. The only time this would be initiated would be when a non-root participant detects that its parent is unresponsive, attempts to connect to a new parent using the protocol discussed in Section 2.1, and receives no response from a higher priority entity.

Upon electing a new root node, the children of former root push all pending requests up to the new root, and the system resumes operation. When a token is believed to be lost by the root node, *e.g.*, after a configurable timeout based on a query of the participants for any outstanding tokens, a new token is generated.

If the root did not die—but believes a token has been lost—it can either multicast a query for the outstanding tokens and regenerate, or it can use targeted messages along the path the token took. In the latter case, the token will either pass normally with a release message or the root will have to regenerate a token. This process is shown in Figure 6. The root participant knows it sent a permission token to D and that it went through child B to get there. There is no reason to send a recovery message to anyone other than B and let it percolate down to D unless multicast or broadcast is being used and the operation is inexpensive for the network.

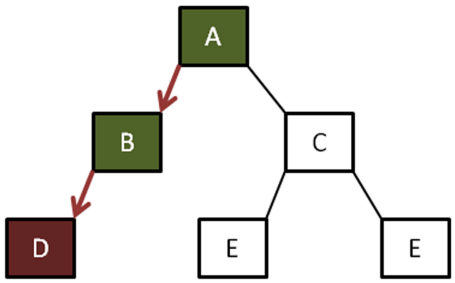

Fig. 6. Targeted Recovery of an Outstanding Token

Any time a parent connection is lost, the orphaned child establishes a new parent with the protocol outlined in Section 2.1 and resends all pending requests from itself and its children.

3 Evaluating the PADME algorithm

This section evaluates results from experiments conducted on a simulated message-oriented prototype of the PADME algorithm over shared memory. We simulate critical section time (the time a participant uses its critical section), message transmission time between participants (the time it takes to send a Request, Reply, or Release message between neighboring cloud participants in the spanning tree), and critical section request frequency (how often a participant will request a critical section if it is not already in a critical section or blocking on a request for a critical section). Our experiments focus on the following goals:

1. **Quantifying the degree of QoS differentiation**. The goal of these experiments is to gauge whether or not the PADME algorithm provides QoS differentiation for participants in a public cloud (see Section 1) and whether or not the Request-Reply-Release models described in Section 2.2 have any tangible effects on QoS differentiation and throughput. Our hypothesis is that the PADME algorithm will provide significant differentiation based on proximity to the root participant.
2. **Measuring critical section throughput**. The goal of these experiments is to measure the critical section throughput of the PADME algorithm. Our hypothesis is that the PADME algorithm will provide nearly optimal critical section throughput for a distributed system, which is the situation where synchronization delay is t_m—the time it takes to deliver one message to another participant.

We created a simulator that allowed us to configure the Priority, Reply, and Release Models for several runs of 360 seconds. The experiments ran on a 2.16

GHZ Intel Core Duo 32 bit processor system with 4 GB RAM. The experiments were conducted on a complete binary tree with seven participants and a depth of 3 on a simulated network of seven participants: one high importance, two medium importance, and four low importance.

3.1 Quantifying the Degree of QoS Differentiation

The QoS Differentiation experiments quantified the ability of the PADME algorithm to differentiate between cloud customers and applications based on their priority—a derived metric that may directly correlate to money paid by the users of the system or a service's importance in serving cloud customers. We present the results as they relate to algorithm configurations and the tradeoffs between reducing priority inversions and increasing file access throughput.

Setup Two experiments are presented here. The first has a message transmit time of 1 sec and a critical section entry time of 1 sec. The second experiment has a transmit time (t_m) of 0.5 sec and a critical section entry time of 1 sec. The latter experiment more accurately emulates network and Internet traffic since transmit time is rarely 1 sec. We use a convention of referencing models as Priority-Request-Reply-Release when describing PADME settings for brevity. Our PADME prototype does not yet include the Replace Request Model, so no

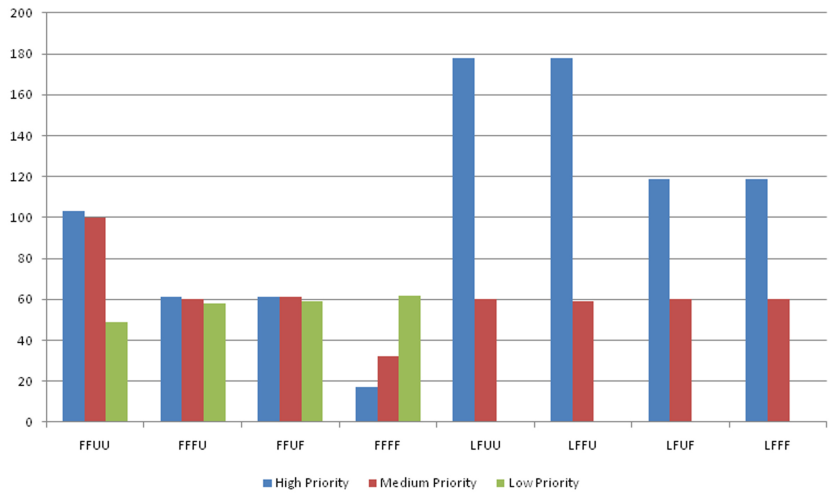

Fig. 7. Priority Differentiation with CS time of 1s and Message latency of 1ms

configurations with the Replace Request Model are included in this section. We expect, however, that throughput and latency for this model should be identical to the Forward Request Model.

Analysis of Results Figure 8 and Figure 7 outline the results for this test. The root participant had high priority, the participants on the second level had medium priority, and the leaf nodes on the third level had low priority.

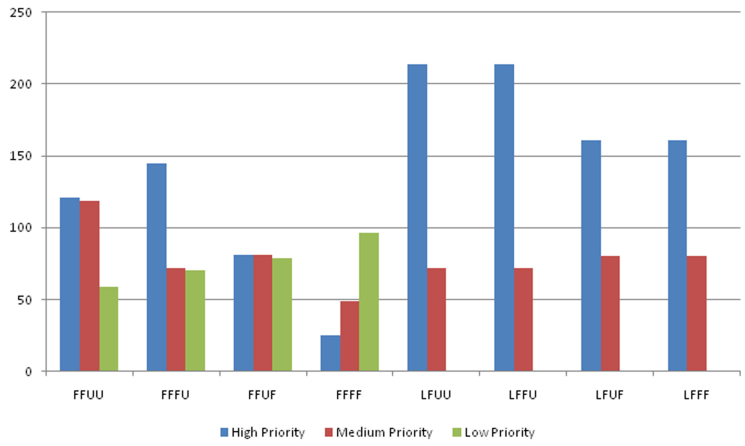

Fig. 8. Priority Differentiation with CS time of 0.5s and Message latency of 0.5ms

In the analysis below we reference the PADME model configurations as Priority-Request-Reply-Release for brevity. Differentiation increases under certain models as the message time is decreased. This result appears to occur in Fair-Forward-Forward-Use, but is likely true of Forward-Use-Forward. Of the Request-Reply-Release combinations that appear to show the best differentiation amongst priority levels, those with Level Priority Model differentiate the best. Those with any type of Level Priority Model differentiate according to priority levels, which makes sense.

More interesting, however, is how the Fair-Forward-Use-Use, Fair-Forward-Forward-Use, and Fair-Forward-Use-Forward model combinations allow for better QoS in comparison to Fair-Forward-Forward-Forward. Although we are being fair in priority policy, this policy shows favoritism to the lower priority levels, which have more participants, and consequently get more critical section entries under a fair priority policy. Forward-Use-Use, Forward-Forward-Use, and Forward-Use-Forward offset these policy decisions by allowing critical section entries as the Reply and Release messages pass through participants, to allow for higher critical section entries than would have been possible with the more intuitive Forward-Forward-Forward. If we increased the number of high priority and medium priority participants, we would have even better differentiation during Fair Priority Policy because less time is spent in pure overhead states where the token is percolating back up the tree and not being used.

3.2 Measuring Critical Section Throughput

Differentiation can be useful, but if the system becomes so bogged down with messaging or algorithm overhead that file system or other shared resource throughput is greatly reduced, then no real benefits are available in the system. The experiments in this section guage the performance of the PADME algorithm in delivering file system access to customers, cloud applications, or persistent services.

Setup Two experiments are presented here. The first experiment sets the message transmission time (t_m) to 1 ms, critical section usage time to 1 s, and we generate a new request once every 1 ms (when not using or blocking on a critical section request). The second experiment has a fast message transmission time of 0.5 ms (*i.e.*, more in line with a local area network transmit for a cluster within a cloud) and generates a new request every 0.5 ms (unless blocking on or using a critical section).

Our PADME prototype does not yet include the Replace Request Model, so no configurations with the Replace Request Model are included in this section. We expect, however, that throughput and latency for this model should be identical to the Forward Request Model.

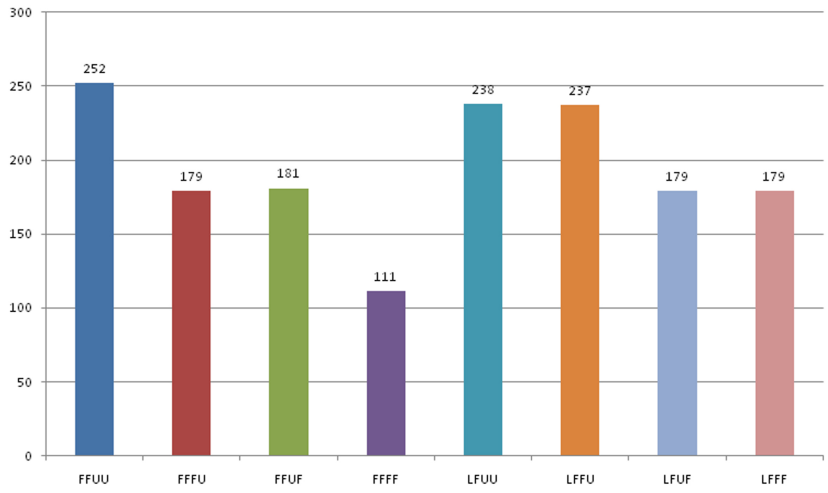

Fig. 9. Throughput with CS time of 1s and Message latency of 1ms

Analysis of Results Figure 9 and 10 show the results for these tests. These results are above our proposed theoretical max where synchronization delay =

t_m, because participants are able to enter their critical sections (*e.g.*, access a file) both on a release and reply using the Use models.

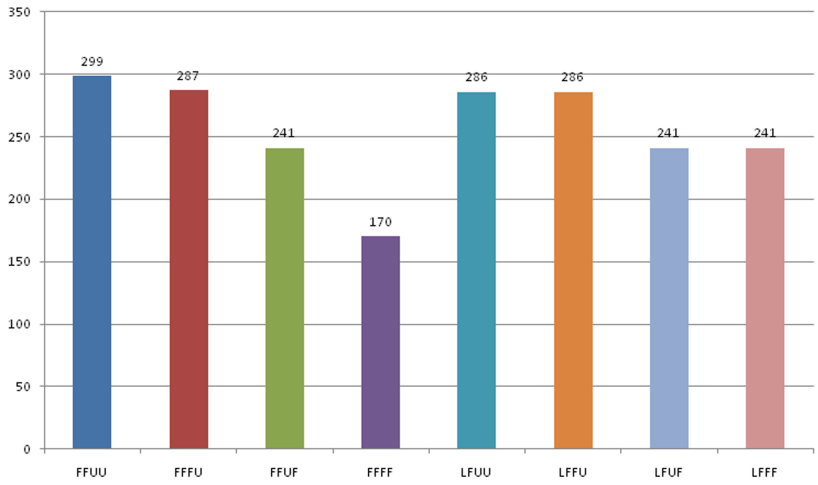

Fig. 10. Throughput with CS time of 0.5s and Message latency of 0.5ms

Each model equals or outperforms a centralized solution. A centralized solution would have required a critical section entry (1 sec) plus two message transmissions—Release (1 sec) and Reply (1 sec)—per access resulting in only 120 critical section entries in a 360s test. The only configuration that performs worse than this one is the Fair-Forward-Forward-Forward combination. A centralized solution would have required a critical section entry (1 sec) plus two message transmissions—Release (0.5 sec) and Reply (0.5 sec)—per access resulting in just 170 critical section entries in a 360 sec test. Every model outperforms or equals a centralized solution in this scenario.

Some settings of the Priority-Request-Reply-Release models allow for the root participant (the highest priority participant) and medium priority participants to enter a critical section twice upon Reply or Release messages. This feature causes an additional critical section entry being possible during Use-Release with a synchronization delay = 0. This result occurs when a new request occurs in cloud applications on the root during or just after the root participant is servicing a separate request.

4 Related Work

This section compares our work on PADME with key types of mutual exclusion solutions in networks, grids, and clouds. We begin with discussions on central token authorities and end with descriptions of distributed techniques in clouds.

A basic form of mutual exclusion is a central authority that delegates resources based on priority or clock-based mechanisms. When a participant needs a shared resource, it sends a request with a priority or local timestamp to this central authority, and the central authority will queue up requests and service them according to some fairness or priority-based scheme. The Google File System (GFS) [2] uses such central authorities (called masters) and has tried to address these issues by creating a master per cell (*i.e.*, cluster). The GFS approach only masks the problems with the centralized model, however, and has a history of scaling problems [1].

The Lithium file system [13] uses a fork-consistency model with partial ordering and access tokens to synchronize writes to file meta data. These access tokens require a primary replica (centralized token generator) to control a branching system for volume ownership. Recovery of access tokens when primary replicas die requires a Byzantine view change of O(n) messages before another access token can be generated, and this can be initiated by almost anyone. In PADME, the view change should only be requested by someone along the token path. When the root node dies with a token still in it, the immediate children of the root would be in the token path and could request a view change. If there are no pending writes or reads, no change may even be necessary, and the root could resolve its fault and continue operations.

Distributed mutual exclusion algorithms have been presented throughout the past five decades and have included token and message passing paridigms. Among the more widely studied early distributed algorithms are Lamport [6] and Ricart-Agrawala [7], which both require $O(n^2)$ messages, and Singhal [8], which uses hotspots and inquire lists to localize mutual exclusion access to processes that frequently enter critical sections. These algorithms are not applicable to cloud computing, where faults are expected. Singhal's algorithm also does not differentiate between priorities since it prefers frequent accessors (which might be free or reduced-payment deployments).

Message complexity has been further reduced via quorum-based approaches. In quorum-based approaches, no central authority exists and the application programmer is responsible for creating sets of participants that must be requested and approved for the critical section to be granted. For the Maekawa quorum scheme to function [9], each set must overlap each other set or it will be possible for multiple participants to be granted a critical section at the same time. If the sets are constructed correctly, each participant has a different quorum set to get permission from, and mutual exclusion is guaranteed. The problem is automatable and involves finding the finite projection plane of N points, but suffers performance and starvation problems (potentially of high priority participant) with faults.

More recently, a distributed mutual exclusion algorithm was presented by Cao et. al. [10]. This algorithm requires consensus voting and has a message complexity of O(n) for normal operation (*i.e.*, no faults). In contrast, our PADME algorithm requires O(d) where d is tree depth—$O(log_b n)$ where b is the branching factor of the spanning tree discussed in Section 3. The Cao et. al. algorithm

also appears to require a fully connected graph to achieve consensus, and does not support configurable settings for emulating many of PADME's QoS modes (such as low response time for high priority participants).

Housni and Trehel [11] presented a grid-specialized token-based distributed mutual exclusion technique that forms logical roots in local cluster trees, which connect to other clusters via routers. Each router maintains a global request queue to try to solve priority conflicts. Bertier et. al. [12] improved upon Housni and Trehel's work by moving the root within local clusters according to the last critical section entry. This improvement, however, could result in additional overhead from competing hot spots, where two or more processes constantly compete for critical sections. Both algorithms treat all nodes and accesses as equals and are susceptible to the problems in Singhal [8] and consequently are non-trivial to implement for a public cloud where paying customers should have differentiation. Moreover, tokens can become trapped in a cluster indefinitely.

5 Concluding Remarks

This paper presented an algorithm called *Prioritizeable Adaptive Distributed Mutual Exclusion* (PADME) that addresses the need to provide differentiation in resource acquisition in distributed computing scenarios. We demonstrated its usefulness in cloud computing environments without requiring a centralized controller. Although we motivated PADME in the context of cloud file systems, it can be used for any shared cloud resource.

The following are lessons learned from the development of this mutual exclusion algorithm:

1. High critical section throughput with differentiation is possible. PADME provides differentiation based on participant priority levels and proximity to the logical root participant of the network. It also provides cloud application developers with four orthogonal models for variety and tighter control of critical section entry. The benefits of the PADME algorithm are high critical section throughput and low synchronization delay between critical section entries, especially when there is high contention for a shared resource.

2. The cost of differentiation is felt by those farthest from the root. In certain settings of the PADME algorithm—especially those using a Level Priority Model—the wait for access to the file system or shared resource could be indefinite. This situation will occur when the cloud is under heavy stress loads, and there is high contention for the shared resource, *e.g.*, when high priority cloud users or applications are constantly accessing the system

3. Fault tolerance should be built-in. Retrofitting fault tolerance into a complex distributed system is hard. The Google File System attempted to solve this with redundant master controllers, but this caused the issue with faulty cloud hardware to just be harder to deal with. Building fault tolerance into a cloud solution from the inception helps reduce time and effort across the lifecycle.

4. Distributed mutual exclusion is still an important topic in Computer Science. We presented results and analysis that show clear differentiation

based on priority level and high critical section throughput. Additional research challenges remain, however, including priority differentiation during tree rebuilding in heavy fault scenarios, reducing message complexity during root consensus changes, virtual overlay networks for individual resources or high priority file system volumes, and secure information flow across the spanning tree to prevent applications or users from snooping file information.

Our ongoing work is empirically evaluating PADME in the context of real public cloud platforms, such as Amazon EC2 and off-the-shelf hypervisors. We are also considering new Request, Reply, and Release Models, as well as more fault tolerance options, to PADME. The open-source PADME algorithm Java code and tests/simulator used in Section 3 are available for download at qosmutex.googlecode.com.

References

1. McKusick, M.K., Quinlan, S.: Gfs: Evolution on fast-forward. Queue **7** (August 2009) 10:10–10:20
2. Ghemawat, S., Gobioff, H., Leung, S.T.: The google file system. In: Proceedings of the nineteenth ACM symposium on Operating systems principles. SOSP '03, New York, NY, USA, ACM (2003) 29–43
3. Ghemawat, S., Gobioff, H., Leung, S.T.: The google file system. SIGOPS Oper. Syst. Rev. **37** (October 2003) 29–43
4. Kshemkalyani, A.D., Singhal, M.: Distributed Computing: Principles, Algorithms, and Systems. 1 edn. Cambridge University Press, New York, NY, USA (2008)
5. Castro, M., Liskov, B.: Practical byzantine fault tolerance. In: Third Symposium on Operating Systems Design and Implementation (OSDI), New Orleans, Louisiana, USENIX Association, Co-sponsored by IEEE TCOS and ACM SIGOPS (February 1999)
6. Lamport, L.: Ti clocks, and the ordering of events in a distributed system. Commun. ACM **21** (July 1978) 558–565
7. Ricart, G., Agrawala, A.K.: An optimal algorithm for mutual exclusion in computer networks. Commun. ACM **24** (January 1981) 9–17
8. Singhal, M.: A dynamic information-structure mutual exclusion algorithm for distributed systems. IEEE Trans. Parallel Distrib. Syst. **3** (January 1992) 121–125
9. Maekawa, M.: A n algorithm for mutual exclusion in decentralized systems. ACM Trans. Comput. Syst. **3** (May 1985) 145–159
10. Cao, J., Zhou, J., Chen, D., Wu, J.: An efficient distributed mutual exclusion algorithm based on relative consensus voting. Parallel and Distributed Processing Symposium, International **1** (2004) 51b
11. Housni, A., Trehel, M.: Distributed mutual exclusion token-permission based by prioritized groups. In: Proceedings of the ACS/IEEE International Conference on Computer Systems and Applications, Washington, DC, USA, IEEE Computer Society (2001) 253–
12. Bertier, M., Arantes, L., Sens, P.: Distributed mutual exclusion algorithms for grid applications: A hierarchical approach. J. Parallel Distrib. Comput. **66** (January 2006) 128–144
13. Hansen, J.G., Jul, E.: Lithium: virtual machine storage for the cloud. In: Proceedings of the 1st ACM symposium on Cloud computing. SoCC '10, New York, NY, USA, ACM (2010) 15–26