Fault-tolerant Quality-of-service-enabled Distributed Mutual Exclusion for Message-Oriented Middleware

James Edmondson, Douglas C. Schmidt, and Aniruddha Gokhale

Vanderbilt University, Nashville, TN USA
{jedmondson, schmidt, gokhale}@dre.vanderbilt.edu

Abstract. Distributed mutual exclusion is the process of ensuring exclusive access to a shared resource between multiple competing threads of execution in a distributed system. Despite the utility of distributed mutual exclusion, conventional message-oriented middleware generally does not support this feature, so application developers who need it must create their own *ad hoc* solutions, which are often inefficient and error-prone. This paper provides two contributions to research on distributed mutual exclusion for message-oriented middleware. First, we describe a quality-of-service (QoS)-enabled distributed algorithm called *Prioritizable Adaptive Distributed Mutual Exclusion* (PADME) that can be implemented in most message-oriented - middleware platforms and which provides high critical section throughput, reduced average synchronization delay, fault tolerance, and priority inversion avoidance. Second, we evaluate the performance of PADME and analyze its QoS and critical section throughput in a representative message-oriented middleware environment. Our results show that the rich feature set of the PADME algorithm addresses a range of application QoS requirements and can reduce synchronization delay to a single message transmission.

Keywords: distributed mutual exclusion, cloud and grid computing, priority differentiation, message-oriented middleware

1 Introduction

Distributed mutual exclusion involves the acquisition and release of shared resources amongst competing distributed participants, which can be a process, component, thread of execution, etc. Once a participant has been granted permission to use this shared resource, it enters its critical section. Many solutions to distributed mutual exclusion have appeared in the research literature (see Section 5 for related work), but few message-oriented middleware platforms support distributed mutual exclusion and even fewer support fault-tolerant, quality-of-service (QoS)-enabled distributed mutual exclusion. When developers need support for distributed mutual exclusion on a shared resource, therefore, they often resort to crafting inefficient *ad hoc* solutions that are centralized (*e.g.,* one static root process receiving all requests for the shared resource and granting individual access) and which provide little to no QoS differentiation (*e.g.,* priority queues of critical section requests and synchronization delay assurance).

There are two primary reasons why distributed mutual exclusion is not (yet) a pervasive middleware feature:

- **Most mutual exclusion problems are naturally expressed using centralized solutions**. For example, achieving mutual exclusion of a shared memory segment on a single computer can be handled via a centralized token authority. A centralized solution running on the computer hosting the shared memory segment incurs significant overhead when the resource is heavily contested, since all mutual exclusion messages are pure messaging overhead and do not contribute to the reading or writing of data from or to the shared memory segment. By moving this messaging overhead to another node, throughput to and from the shared memory segment is likely to speed up. A distributed mutual exclusion scheme often offloads this overhead to other nodes, participants, or threads of execution in the networked system [14].

- **Implementation complexity**. Application developers tend to create *ad hoc* centralized solutions because there are few readily available implementations of distributed mutual exclusion algorithms and distributed algorithms are hard to design, optimize, and debug. What is needed, therefore, is a robust, general-purpose algorithm that can be integrated readily into popular message-oriented middleware platforms, such as MPI [3], DDS [12], or Real-time-CORBA [13].

This paper presents an algorithm called *Prioritizable Adaptive Distributed Mutual Exclusion* (PADME) and techniques to implement it in message-oriented middleware. PADME is designed to alleviate the drawbacks with *ad hoc* mutual exclusion approaches and to provide middleware developers with a robust, general-purpose algorithm that provides distributed mutual exclusion along with configurable models to ensure high critical section throughput performance and scalability, as well as avoiding priority inversions and tolerating participant failures. PADME is based on a topological tree technique that supports fault tolerant mutual exclusion, quality-of-service (QoS) differentiation amongst participants in the network, and increased performance for high priority participants. In addition, PADME optimizes performance via flexible model variations that reduce *synchronization delay* (synchronization delay) (the time between a participant leaving its critical section and the next participant entering its own critical section) to a single message transmission, which is a significant improvement over traditional algorithms (See Section 5) and scales well as workload increases. The PADME algorithm also improves *critical section throughput* (*i.e.,* the number of critical section entries over a particular period of time).

The remainder of this paper is organized as follows: Section 2 outlines a motivating scenario that requires QoS-enabled distributed mutual exclusion; Section 3 describes the PADME algorithm and shows how it can be implemented efficiently on message-oriented middleware and cloud/grid platforms; Section 4 evaluates results from experiments conducted on a representative message-oriented middleware implementation of this algorithm; Section 5 compares our approach with related work; and Section 6 presents concluding remarks.

2 Motivating Scenario

Imagine a search and rescue scenario shown in Figure 1 where autonomous robotic agents have been deployed into a devastated area, *e.g.*, due to an earthquake, flood, hurricane, etc. These agents have been designed to search for and detect human survivors. In an ideal scenario, each agent would have unlimited communication resources available to them, but in reality the disaster may have knocked out most communication infrastructure or there may be information overload due to many deployed sensors and personnel competing for scarce computing and networking resources.

To make this example more concrete, the robotic agents and the environment they operate in include the following capabilities:

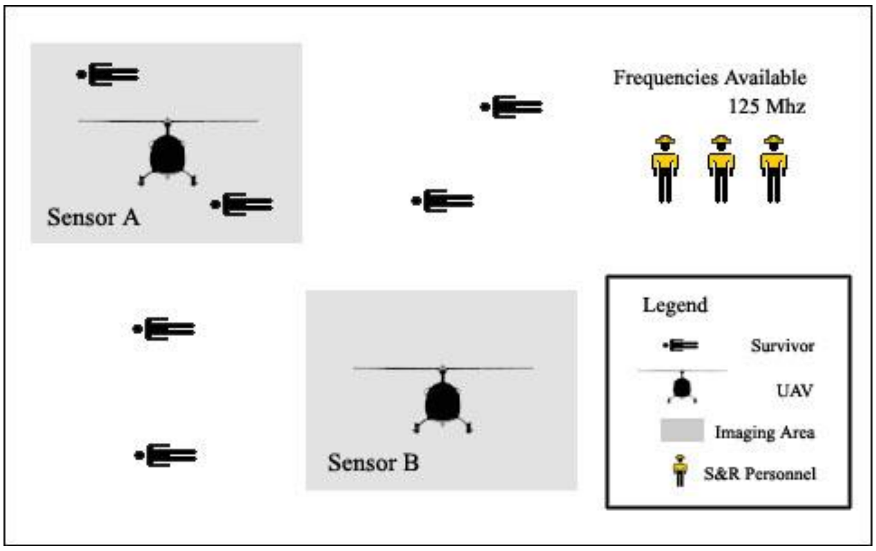

Fig. 1. Search and Rescue Scenario Where Rescuers are Given a Shared Resource (Frequency) to Receive Video of Survivors.

- A network communication medium that allows agents to communicate with each other. A likely candidate for this type of communication would be a short wave radio that has a finite number of frequencies, many of which are reserved for emergency channels that these agents must not interfere with to avoid conflicts with other rescue operations in the area. The available frequencies are thus a scarce, important resource that needs mechanisms for sharing them via distributed mutual exclusion and to ensure that the most important information possible is being relayed across them.
- The ability to detect human presence within 50 yards, *e.g.*, based on infrared signatures, audio sensors, etc. The number of humans detected within range of a robot should elevate the priority of this autonomous agent's information. In other words, the more humans detected still alive in this sector, the more important this agent's information, and hence this agent must receive preference over other agents to acquire the shared resources, such as the short wave frequencies.
- Built-in cameras or other data collection sensors that might be useful to rescue workers. A camera could give the rescue team reference points for finding people in need of help. It could give the rescue team information about how stable the environment around the trapped persons might be (*e.g.* heavy structural damage). GPS and other types of information may also be valuable and if able to be collected, may give rescue workers even more information.
- The ability to transmit across the network communication medium for a set period of time (called *critical section entry time*) and then cut off transmitting video or other data until they gain access to the distributed critical section once more. This ability will allow other equally important agents to transmit their video feeds or data for a period of time as well, without one agent using the transmission medium indefinitely.

- Potential interruption of communication via obstacles, distance, etc. We therefore need ability to handle participants of this scenario joining and leaving the communication.

An appropriate solution to this type of problem should address the following requirements and challenges:

- **Challenge 1**: **Prioritized mutual exclusion based on the importance of information being disseminated by a particular agent**. Agents with large numbers of humans in their areas of interest should be able to lock the available frequencies for video or data transmission more often than agents with no or fewer humans detected. Moreover, we should try to reduce priority inversions (*i.e.*, always prefer video of survivors over videos without survivors). An ideal solution would be flexible enough to change the priority mechanisms that determine who is a more important agent later on or in different scenarios. For example, after an initial scan of all sectors is completed and rescue personnel have found hotspots (*e.g.*, the most crucial places) of rescue need, the rescue workers may want to change the priority of the agents in the field to a more fairness-based strategy, so that views of all sectors may be seen in a round-robin manner. Section 3.2 describes how the PADME algorithm supports prioritized mutual exclusion by providing models that support both priority level differentiation and fairness with additional models that result in fewer priority inversions than a traditional centralized scheme.
- **Challenge 2: Fault tolerance with respect to obstructions or equipment failure.** The algorithm for mutual exclusion of frequencies should be robust against agent failure or issues with line-of-sight obstructing short wave radio transmissions. Section 3.3 describes how the PADME algorithm supports fault tolerance by reducing message complexity required to bring new participants or threads of execution up to speed.
- **Challenge 3: Maintaining high critical section throughput and low message complexity, despite resolving Challenge 1 and Challenge 2.** The communication required for mutual exclusion (message complexity) should be minimized, *i.e.* the number of messages required for mutual exclusion should be as low as possible since the reserved frequencies for background traffic will likely be just as scarce as frequencies available for data traffic. A separate part of the performance challenge is maximizing the *critical section throughput* (the number of critical section entries that occur during a given period of time). A key component of critical section throughput is the minimization of *synchronization delay*, which is the downtime that a mutual exclusion algorithm experiences between one thread leaving its critical section and the next thread entering. Minimizing synchronization delay helps to maximize critical section throughput. Section 3.4 analyzes each of PADME's configurations and shows how the different PADME models can be used together to result in reduced average synchronization delay and improved performance as more critical section entries are requested.

A wide range of underlying middleware platforms can be used to support the search and rescue mission. For example, agents can be implemented using cloud technologies, such as Apache Hadoop [2], or communicate with grid computing middleware, such as MPICH_2 [3]. To accomplish a distributed mutual exclusion algorithm suited for any scenario, however, the solution should be general-purpose, reusable, and support key QoS properties, such as fault tolerance (in case participants go down or lose communication periodically), avoid priority inversions, and provide high critical section throughput. The PADME algorithm and middleware address these requirements.

3 The PADME Algorithm and Its Manifestation in Middleware

This section presents an algorithm called *Prioritizable Adaptive Distributed Mutual Exclusion* (PADME) that we developed to meet the challenges described in Section 2. The PADME algorithm requires a user or middleware provider to conduct two preparatory operations:

- Building a spanning tree of the participants in the network. The spanning tree needs a logical root node that acts as a token authority to which permission for mutual exclusion eventually returns. If the spanning tree is a binary tree or m-way tree[1] this logical token authority would be the root of the tree (and this would be the most efficient selection).
- Selecting preferred models 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 the root node and a requesting node). Each model may be changed during runtime if required by the middleware or users.

The remainder of this section describes the PADME algorithm and shows how it can be implemented efficiently on message-oriented middleware.

3.1 Building the Logical Spanning Tree

The logical spanning tree is built by informing a participant of its parent. A participant does not need to be informed of its children, as they will eventually try to contact their parent, establishing connections on-demand. We use this same mechanism to reorder the tree when we are trying to optimize certain high priority participants. It is each participant's responsibility to reconnect to its parent.

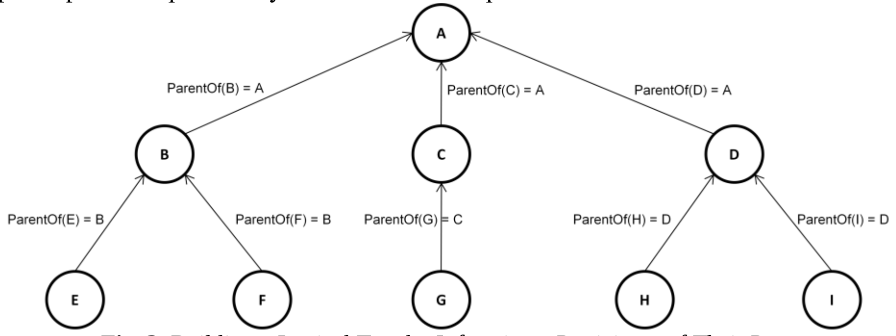

Fig. 2. Building a Logical Tree by Informing a Participant of Their Parent.

Figure 2 shows the construction of such a spanning tree. During runtime, an application or user may add or remove a participant, rename participants, or conduct other

[1] Though this paper refers to the spanning tree as a binary tree or an m-way tree, the PADME algorithm can support any type of spanning tree.

such operations to organize our intended tree and dynamically respond to changes in request load or priority changes. The good news is that this just requires updating affected participants with parent information (*i.e.* informing them which direction the logical root of the tree is at). Obviously, if the middleware only supports static assignment of ranks (as is the case with MPI), then this functionality can be unimplemented or unutilized in the middleware, but for more dynamic middleware, such as DDS [12] or cloud computing technologies like Hadoop [2], the ability to add or remove participants is important.

In general, higher priority participants should be moved closer to the root of the logical tree. Moving participants closer to the root yields lower message complexity, faster synchronization delay, better throughput, and higher QoS for the target system, as shown in Section 3.4.

3.2 Models and Algorithm for Distributed Mutual Exclusion

The basic model of the PADME algorithm requires just three types of messages: Request, Reply or Grant, and Release. A Request message is made by a participant that wants to acquire a shared resource, such as the short wave frequency. 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 algorithm 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 or not an intermediate participant will be allowed to enter its own critical section upon receipt of the message type. These models are not a result of our motivating scenario, but are a consequence of our approach to distributed mutual exclusion and optimizations that can be made to allow shorter synchronization delay (synchronization delay) between critical section entries and improved QoS as a part of user-specified requirements to middleware.

The configurations of the Request, Reply, and Release models may be changed at run time to result in different QoS, including higher critical section throughput (*i.e.*, more critical section entries over time), 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), less priority inversions (in this context priority inversions refer to 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.

Request Models. There are 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 parent 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. There are 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. Use Reply Model results in higher critical section throughput, lower synchronization delay, and more priority participants being serviced despite using a Fair Priority Model, described later. Note that this model also affects the root node, which may enter its critical section before sending a Reply message to the appropriate child if the Use Reply Model is enabled. An additional note about the Use Reply Model is that the participant should not send a Release message for itself until the intended requester participant has sent its Release back up the chain. At that time, the participant that took advantage of the Use Reply Model should append its identifier onto the Release message and send it on.

Release Models. There are 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 or not it has a request pending. The Use Release Model allows a participant to enter its critical section upon receiving a Release message from one of its children, if the participant has an outstanding Request pending. Note that this also affects the root node, which may enter its critical section upon receiving a Release message before servicing the next Request, if Use Release Model is enabled. An additional note about Use Release Model is that the participant will need to append its identifier onto the Release message if it entered its critical section (see Figure 3), and that this may result in a Release message containing multiple instances of the participant identifier in the Release message. Consequently, the identifiers in a Release message may not be held in a Set but instead a Multiset, to allow for duplicates of the same identifier. These duplicates will allow for proper bookkeeping along the token path, since up to two Request messages may need to be removed from each affected priority queue.

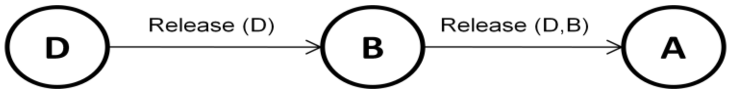

Fig. 3. Release Chaining when using a Use Reply Model or Use Release Model. Each participant appends release information from themselves to their parents (when the critical sections have already been entered)

Priority Models. There are 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, then 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 and then 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), while the Level Priority Model makes no such guarantees.

The PADME algorithm for mutual exclusion. When a participant (in this paper, we refer to a participant as an individual processing element potentially interested in mutual exclusion on a shared resource or simply a processing element that takes part in the routing of messages – it does not refer to a threading model) needs to enter its critical section (*e.g.* an agent is requesting exclusive access to a frequency for broadcasting), 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 *e.g.*user data structure that indicates the shared resource id (*e.g.* the frequency in the motivating scenario) and any other data relevant to business logic. There is no reason that any of these variables be limited to integers only. For the purpose of brevity, we will not go over distributed election of identifiers, also called the Renaming Problem [9].

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 3.4. We recommend using either a timer that is updated (1) only when sending a Request or (2) any time a Request, Reply, Release, or Sync 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, a few of which are described in Section 3.3. 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 frequency that has been granted. 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, … I_n> or <I_0, D_0, I_1, D_1, … I_n, D_n> where I_0, I_1, … I_n is a list of participant identifiers that used their critical section along this token path, and D_0, D_1, … D_n 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, … I_n before forwarding the Release to its parent for proper bookkeeping. The process of sending a Request, Reply, and Release message is shown in Figure 4.

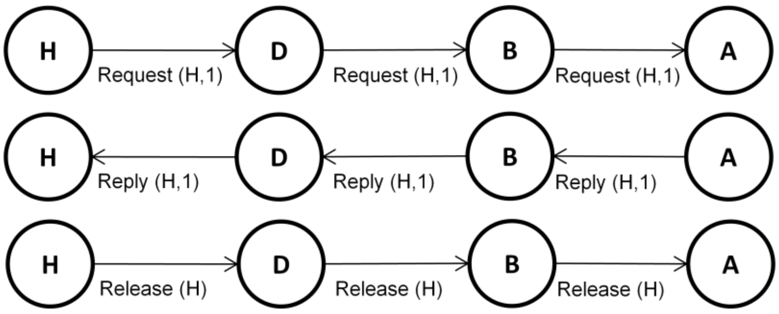

Fig. 4. Messaging Required for a Critical Section Entry from Participant H to a Logical Root Participant A (Diagram Assumes a Forward-Forward-Forward Configuration).

3.3 QoS Properties of the PADME Algorithm

The Request, Reply, Release, and Priority Models described in Section 3.2 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 need better than the others, *e.g.*, each has certain synchronization delay characteristics, throughput, and even message complexity differences during fault tolerance. Synchronization delay is the time between some participant leaving a critical section and the next participant entering it and is a component of critical section throughput. To simplify understanding the different combinations of these models, we created a system of model combinations called Request-Grant-Release that codify these combinations.

The most robust Request-Reply-Release model combination is the Replace-Use-Use model, which corresponds to Replace Request Model, Use Reply Model, and Use Release Model. The Replace-Use-Use combination requires each participant to keep a priority queue for child Requests (described further in Section 3.2 information), but to summarize, 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. The Use Reply Model of Replace-Use-Use 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 should 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 3.4 analyzes message complexity and synchronization delay).

Another key QoS benefit of using the PADME algorithm is that fault tolerance is simplified. For example, only one new message (the Sync message) must be introduced to allow for seamless operation during faulty periods, and an example of when this is required is shown in Figure 5.

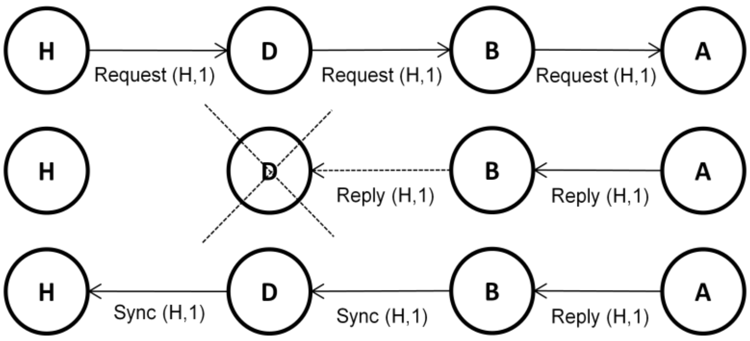

Fig. 5. Example Situation that Requires a Sync Message from B to H.

Below we describe the various fault conditions and how participants should deal with these situations. When a parent participant dies the following steps occur:

- Child resends Requests according to its Request Model
- If using Replace, only the most important Request is resent. If the parent dies again, resend an additional time

When a child participant dies the following step occurs (example of which is shown in Figure 4):

- If a token has gone down a path (*i.e.* a Reply has gone through it to some target) that included that child, send a Sync <I, C> message where I is the recorded id of the initiating requestor and C is the clock or count of the Request. Note that a Reply message can serve as a Sync message as they serve similar functions (*i.e.* a middleware implementer may choose to resend the Reply instead of implementing a Sync message). Semantically, a Sync message is a reminder of a previously sent Reply message along the path to the initial requestor.

When a participant receives a Sync <I, C> message where P_i == I

- If P_i still needs the critical section for this count (*i.e.* it never received the initial Reply due to an ancestor participant failure), then it enters its critical section
- P_i sends a Release <P_i> message to its parent

We do not discuss at length how a participant might detect a parent or child faulting or dying, and for the most part, we leave this to the implementer. If using TCP channels between participants, for instance, this could be determined whenever a connection is lost. If using a connectionless protocol (*e.g.*, UDP), a type of heartbeat (recurring message to establish liveliness) may be used to determine when resends or Sync messages are needed.

3.4 Analysis of the PADME Algorithm

We now briefly analyze key Request-Grant-Release combinations for best and worst case performance of the PADME algorithm. We do not consider the effects of faults on performance in this analysis. For empirical evaluation of the PADME algorithm, see Section 4.

For all Request-Reply/Grant-Release combinations and Priority Models, critical section (critical section) message complexity ranges from O(0) in the root participant, since it does not have to send a message to itself, to O(3d) for all depths that are reachable in the tree. In Fair Priority Model, d is equal to the maximum depth of the tree (d_m), and eventually a $3d_m$ message complexity critical section is reached, since under a fair scheme, all leaf nodes will eventually be serviced, and the deepest leaf will require $3d_m$ messages. Since dm = $\log_b n$, where b is the branching factor of the tree, this turns out to be a manageable worst case. If our solution is used in something like a token ring, a Fair Priority Model may be prohibitive, depending on performance requirements, since such a topology would require O(n) messaging.

Any Level Priority Model combination, in contrast, will have message complexity per critical section entry not exceeding O(3d) where d = d_s. d_s is the maximum depth of a serviced participant. If high priority participants are constantly competing against each other in a level priority scheme, no lower priority participants at lower levels will ever enter their critical section. In Level Priority Model, this often means that if the children of root are constantly requesting critical section entry, message complexity never exceeds O(3) for critical section entry (a Request, Reply, and Release sent from child of root to root directly) and synchronization delay (synchronization delay) is reduced to O($2t_m$), since assuming more than one child of root is constantly requesting entry, as soon as root receives a Reply message, it need only send a Release message to the next candidate before another critical section entry is accomplished. No combination of Use Reply Model or Use Release Model can result in a child of a participant at d_s ever entering a critical section in a Level Priority Model. Proof of this is left as an exercise to the reader. The only situation where $d_s = d_m$ for Level Priority Model is in an underutilized system (*e.g.* low request rates), where leaf nodes periodically get to enter critical section regions because there is simply no higher priority request pending. Otherwise, d_s is less than d_m.

The most interesting Request-Grant-Release combination is Replace-Use-Use, used in concert with the Fair Priority Model. This combination results in all participants eventually entering a critical section (fairness property) and can also result in synchronization delay being reduced to just t_m when each participant along a token path needs to enter a critical section. This means that not only do we guarantee all participants will eventually enter a critical section, but under heavy loads with all participants constantly requesting, we achieve minimal synchronization delay for a distributed system, since all such systems require at least one message be sent to inform another participant of an available critical section.

The analyses of synchronization delay and critical section throughput for Replace-Use-Forward, Replace-Forward-Use, Forward-Use-Forward and Forward-Forward-Use are similar, so we lump them together here. Analysis of Level Priority Model with these mechanisms is trivial if children of root are constantly requesting. Under heavy loads seeing continuous requests made by children of root, users could expect synchronization delay of O($2t_m$) and message complexity for critical section entries at O(3). In Fair Priority Model, again assuming all participants continuously requesting unless in a critical section, message complexity rises to O($3d_m$), which is the same as Forward-Forward-Forward with Fair Priority Model, but synchronization delay is reduced from O($2d_m t_m$), which is the time to send a Release from a leaf node to root and a Reply to a subsequent leaf node Request to O($d_m t_m$). This O($d_m t_m$) again assumes heavy load and the source of this overhead depends on which model of Use is represented in the Request-Grant-Release combination. If a Use Release Model is used the O($d_m t_m$) results from a Reply possibly going from root to a leaf node. On the way back up from the awarded leaf node, synchronization delay is reduced to O(tm) as each Release will result in a critical section entry (assuming heavy load and all participants requesting when they are not in a critical section).

Our analysis above shows that under heavy usage, the PADME algorithm should perform as well as any other distributed mutual exclusion solution, often reducing synchronization delay to just t_m and ensuring high critical section throughput during peak usage. Section 4 validates these assumptions via empirical tests and further analysis.

3.5 Integrating the PADME Algorithm into Message-Oriented Middleware

Ideally, a middleware platform should present users with a concise interface to the mutual exclusion mechanisms it supports. Table 1 outlines the application programming interfaces (APIs) we developed so that applications can use the PADME algorithm's mutual exclusion operations on conventional message-oriented middleware, such as MPI.

Table 1. API for Mapping the PADME Algorithm into Message-Oriented Middleware

Function	Description
CriticalSectionSetup(PriorityModel, RequestModel, ReplyModel, ReleaseModel)	Setup the Priority, Request, Reply, and Release models. Each parameter could be an enumerated integer type, classes derived from a super class, etc..
EnterCriticalSection()	Block on a critical section
EnterCriticalSection(UserData D)	Block on a critical section on a particular user descriptor D
LeaveCriticalSection()	Leave the critical section

For each of these entry points into the middleware, we do not require applications to have knowledge of their identifiers in the routing network or the timer. The only information required from application is the desired model setup and any user-specific data (*e.g.*, the frequency being requested, if applicable).

The PADME algorithm can be integrated with a range of middleware architectures. For example, PADME can be implemented in MPI by mapping its distributed mutual exclusion interfaces to a blocking send operation already incorporated since MPI_CH1 [3]. When the user requests a critical section entry, the participant would block on the operation until it gained appropriate permission, and then the user application would continue. Ideally, this implementation would not operate like a MPI_Broadcast or MPI_Barrier (which require every participant to make the function call), but the MPI standard may require it to act in such a way since implementations are not required to have a dedicated thread to process messages unrelated to user application logic. An alterantie approach would incorporate the distributed mutual exclusion processing loop into the MPI multiprocessing daemon layer that facilitates MPI_Process discovery and loads MPI programs.

It may not be readily apparent how to interface other message-oriented middleware, such as DDS [12], and distributed object computing middleware, such as Real-time CORBA [13]. For DDS, we recommend the UserData structure be split into at least a topic (a feature of publish/subscribe paradigms in which publishers publish data to a topic and subscribers subscribe to events on the topic) and a user specified id (string or integer). The latter id would potentially refer to a specific resource or a catch all (for the first available resource of the type). In the context of our motivating scenario in Section 2, this id could represent the frequency to lock when disseminating video or images.

For QoS-enabled distributed object computing middleware, such as Real-time CORBA, the most natural incorporation for a distributed mutual exclusion algorithm like PADME would be to mirror the current semantics of mutual exclusion between thread pools and threads via the `orb->create_mutex()` call by creating a new distributed mutex class that inherits from a RT mutex class. A parameter could be passed to the `create_mutex`() function, indicating that it be invoked across a group of networked threads or participants, rather than local thread pools. Other function calls, such as `set_ceiling()` could be supported to allow multiple enumerated shared resource ids (*e.g.*, id 0-3 to provide for 4 frequency identifiers available to the search and rescue crews and UAVs shown in Section 2). For further usability, the functions shown in Table 1 could be wrapped into scoped locks so methods could seamlessly enter and leave distributed mutual exclusion regions without having to explicitly state EnterCriticalSection and LeaveCriticalSection calls (since scoped locks perform these automatically in their respective constructors and destructors).

4 Empirical Evaluation of the PADME Algorithm

This section evaluates results from experiments conducted on a simulated message-oriented implementation 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, and the 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 results are separated into two groups:

- **QoS differentiation**. The goal of these experiments is to gauge whether or not the PADME algorithm provides QoS differentiation for participants required in the Motivating Scenario (Section 2) and whether or not the Request-Grant-Release models described in Section 3.2 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.
- **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.

The only way to reduce synchronization delay below t_m is to have critical section entries that require no messages. Although this reduction is possible by constructing test scenarios or implementations of distributed algorithms that never allow tokens to leave a single participant, this is not an optimal *distributed* mutex. For a mutex to be distributed, critical section entries must be granted to more than one participant, which requires at the very least a message transmission, which requires at least 1 message transmit time t_m.

We created a simulator that allowed us to configure the Priority, Reply, and Release Models for several runs of 360 seconds. The simulator infrastructure removes much of the possible human error by providing timers that allow automated start and shut off of the experiments at user defined times. All experiments ran on a 2.16 GHZ Intel Core Duo 32 bit processor system with 4 GB RAM. Experiments were conducted on a complete binary tree with 7 participants and a depth of 3. All experiments were conducted on a simulated network of seven participants: one high importance, two medium importance, and four low importance. All the simulator and test code is available on the PADME project website [6].

4.1 Experiment 1 – QoS Differentiation

Setup. Two experiments are presented here. The first has a message transmit time of 1s and a critical section entry time of 1s. The second experiment has a transmit time (t_m) of .5s and a critical section entry time of 1s. The latter experiment more accurately emulates network and Internet traffic since transmit time is rarely 1s.

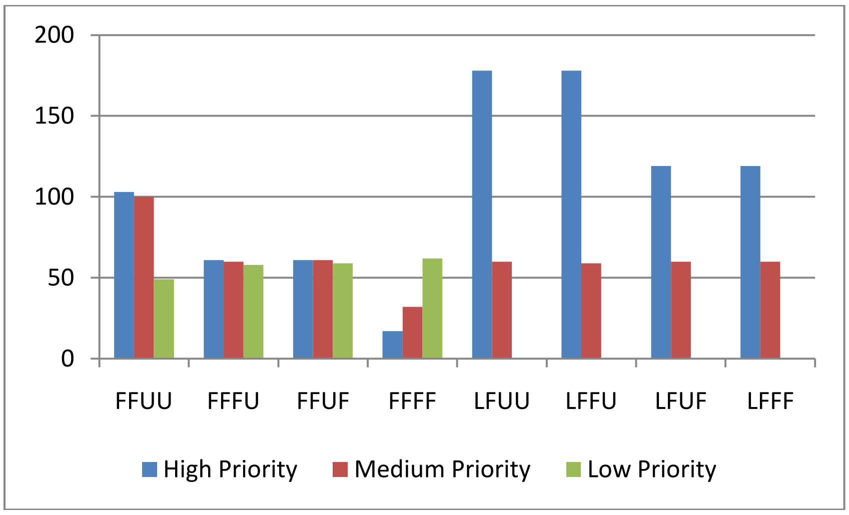

Fig. 6. QoS Differentiation in Solution with t_m = 1s and critical section time = 1s. Horizontal access shows the Priority-Request-Reply-Release models that were being evaluated. Vertical axis indicates number of mutexes entered.

Results. Figure 6 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. The abnormalities seen in the Level-Forward-Use-Forward (LFUF in the figures) results are caused by an implementation detail. Each participant generates a new request at the end of its timer loop, which caused some issues when using the Use Reply Model. The timing loop would process a Request message first, and then Reply before generating a Request for the root participant for that time slice. This sequence of events resulted in not using a critical section, even though one was scheduled to occur at the time instance when a Reply to a Request was being sent. If this request code were moved to the top of the loop, the differentiation between Level-Forward-Use-Forward and Level-Forward-Forward-Use should be similar.

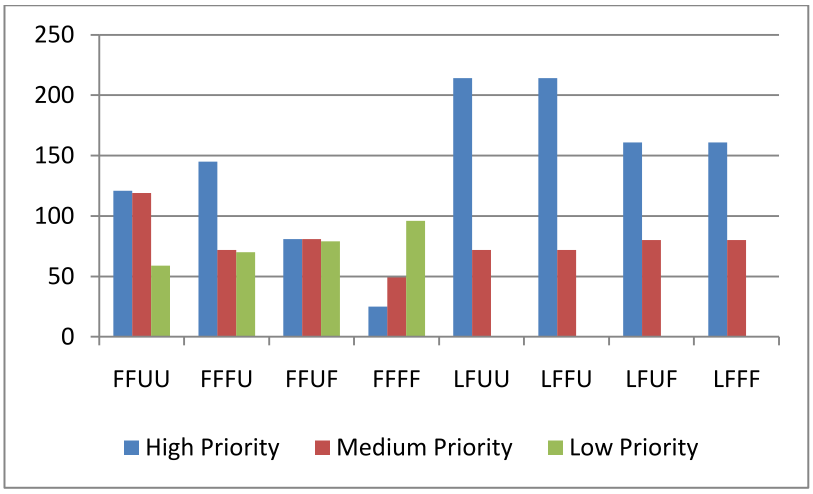

Fig. 7. QoS Differentiation in Solution with t_m = .5s and critical section time = 1s. Horizontal access shows the Priority-Request-Reply-Release models that were being evaluated. Vertical axis indicates number of mutexes entered.

Analysis of results. 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 when ran with the implementation changes noted in the Results section of this experiment. Of the Request-Grant-Release combinations that appear to show the best differentiation amongst priority levels, those with Level Priority Model appear to differentiate the best, and those with any type of Level Priority Model differentiate well in general, 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. Even though 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 would have increased the number of high priority and medium priority participants, we would have even better differentiation during Fair Priority Policy.

4.2 Experiment 2 – Critical Section Throughput

Setup. Two experiments are presented here. In the first experiment, we set the message transmission time (t_m) to 1ms, critical section usage time to 1s, and we generate a new request once every 1ms (when not using or blocking on a critical section request). The second experiment has a fastemessage transmission time of .5ms and generates a new request every .5ms (unless blocking on or using a critical section).

Results. Fig 8 and 9 outline the results for these tests. These results were unexpectedly above our proposed theoretical max where synchronization delay = t_m, but we were able to trace the source of this error to an implementation detail. The issue comes from the root participant generating a new request every .5s and consequently being able to field 2 back to back requests with synchronization delay = 0, skewing the results past our proposed theoretical maximum.

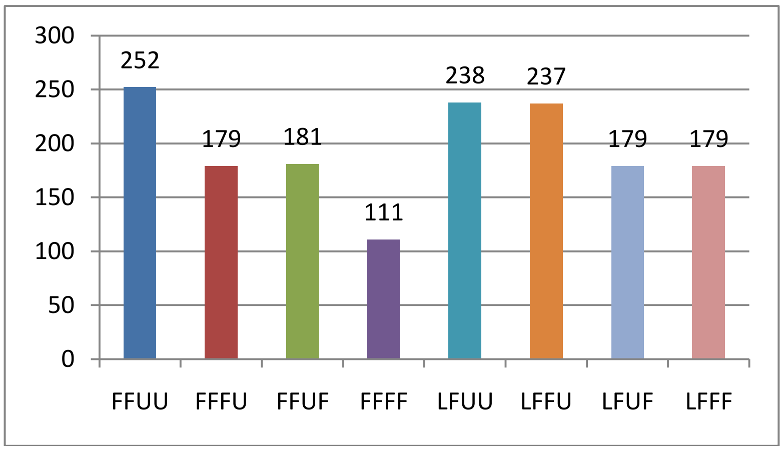

Fig. 8. Total critical section entries in solution with t_m = 1s and critical section time = 1s during a 360s test. Horizontal access shows the Priority-Request-Reply-Release models that were being evaluated. Vertical axis indicates number of mutexes entered.

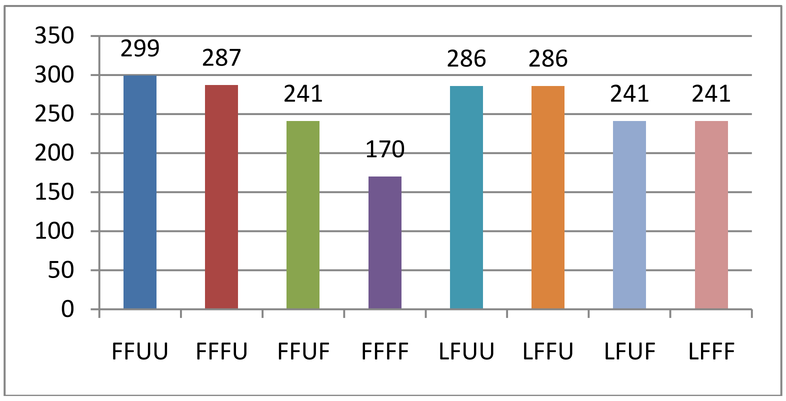

Fig. 9. Total critical section entries in solution with t_m = .5s and critical section time = 1s during a 360s test. Horizontal access shows the Priority-Request-Reply-Release models that were being evaluated. Vertical axis indicates number of mutexes entered.

Analysis of results. Each model equals or outperforms a centralized solution. A centralized solution would have required a critical section entry (1s) plus two message transmissions – Release (1s) and Reply (1s) - per access resulting in just 120 critical section entries in a 360s test. Note that the only configuration that performs worse than this is the Fair-Forward-Forward-Forward combination. A centralized solution would have required a critical section entry (1s) plus two message transmissions – Release (.5s) and Reply (.5s) - per access resulting in just 170 critical section entries in a 360s test. Every model outperforms or equals a centralized solution in this scenario. In fact, the PADME algorithm often beats the "optimal" solution [9] based on synchronization delay being one message (1s in the first test and .5s in the second test) which results in 180 and 240 critical section entries, respectively.

This last accomplishment is partly due to implementation details which allow for the root participant (the highest priority participant) and medium priority participants being able to enter a critical section twice during Use-Release models since our software agents run a single loop for event generation and consumption. This "feature" results in an additional critical section entry being possible during Use-Release with a synchronization delay = 0.

If we were to correct this issue, any models above the optimal critical section entry levels of 180 and 240, respectively, would revert to the optimal critical section entry levels of 180 and 240. We leave the implementation as is for two reasons. First, to show that middleware implementers can use our solution to push past the observed optimal distributed critical section throughput level by effectively reducing high priority participant synchronization delay to 0, where applicable (*e.g.*, another critical section request is available right after a participant finishes a critical section entry during a Use-Release model). Second, this implementation feature/bug was found post facto after analyzing results, was not intentional, and the effects of doing this in the implementation were predictable.

5 Related Work

This section compares our work on the PADME algorithm with key types of mutual exclusion solutions in research literature. A basic form of mutual exclusion is a central authority that delegates resources based on priority or clock based mechanisms. When a participant is in need, it will send a request with a priority or local time stamp to this central authority, and the central authority will queue up requests and service them according to some fairness or priority based scheme. Requests are serviced and access is granted by a simple message back to the winning requestor. The simplicity of this centralized model, along with its power and flexibility, in many ways inspired the solution that we have included in this paper.

The issues with a central authority are well documented throughout research literature and include, but are not limited to, the central authority acting as a message chokepoint, trust issues, faulty central authorities causing retransmission of up to N requests (since it is the only participant storing requests), etc. Consequently, mutual exclusion research in the past few decades has focused on distributed mutual exclusion algorithms – algorithms that do not require a central authority that emulates a token passing mechanism for participants to enter their critical sections [9].

In token based schemes, one or more tokens are held by participants that have been granted access to a resource. For a participant to be granted a critical section, it first requests access from all participants. If it receives success messages, it enters its critical section and signals to other participants that they may proceed. Most of these algorithms tend to require O(n^2) messages under heavy load conditions, and examples of these types of algorithms include Lamport's time based protocol [10] and the modifications proposed by Ricart-Agrawala [15]. Other intelligent schemes like Singhal's solution involving hotspots and inquire lists [16] tend to target requests to participants that have been known to make requests before, causing reduced average message complexity of O(n) – a significant improvement.

Though token based approaches tend to be the easiest to code and often the most prolific implementations, message complexity has been further reduced via quorum-based approaches. In this algorithm, no central authority exists and the application programmer is responsible for creating √n sets of participants that must be requested and approved for the critical section to be granted. For the Maekawa quorum scheme to function [8], 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. Maekawa showed that finding optimal sets for usage by the mutual exclusion algorithm is equivalent to finding a finite projective plane of N points and is non-trivial. For a network with participants entering and leaving constantly, constructing such sets can be a problem.

A separate quorum-based technique was submitted by Agrawal [1] and it mirrors the topology and much of the methodology of the solution we present. In this algorithm, a network topology (*e.g.* tree, grid, binary tree, etc.) organizes the participants logically and a routing mechanism allows for all participants to communicate together (if it is not a fully connected network of participants). In a binary tree based version, quorums are constructed from leaf nodes to some common root node (node 1 in their paper), and all participants along that path (from leaf to root) must get permission from each participant along that path – a result of log (n) messages required for all critical section entries.

More recently, a distributed mutual exclusion algorithm was presented by Cao et. al. [5]. This algorithm requires a consensus voting and has a message complexity of O(n). In contrast, our PADME algorithm only requires O(log n). 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 [8] presented a 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 deconflict priority conflicts. Bertier et. al. [4] 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 simply result in additional overhead from competing "hot spots," where two or more that constantly compete for critical sections.

Middleware platforms incorporating distributed mutual exclusion include a general distributed algorithm selection and scheduling framework called Algon [14]. This framework sits atop a middleware layer and interfaces to applications via a heavy-weight component. The components are organized by a scheduler and a selected algorithm, and each of the components communicates via Java Remote Method Invocation or other similar middleware (paper mentions CORBA IIOP, DCOM or .NET). The distributed mutual exclusion algorithm supported by Algon is Ricart-Agrawala [15] (which as noted above requires O(n^2) messages per critical section entry). PADME can be incorported into the Algon mutual exclusion algorithm selection list to improve application QoS.

6 Concluding Remarks

This paper presented a distributed mutual exclusion algorithm called *Prioritizeable Adaptive Distributed Mutual Exclusion* (PADME). PADME provides differentiation based on participant priority levels and proximity to the logical root participant of the network. It also provides distributed 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 under many of the models, especially when there is high contention for a shared resource.

We also described interfaces and potential entry points into conventional message-oriented middleware platforms to help facilitate integration. In addition, we presented results and analysis for each of the configurations of the PADME algorithm Reply, Release, and Priority models and show how these results tie into a search and rescue scenario. These results show clear differentiation based on priority level and high critical section throughput, which may be changed as latency, individual process priority, or other system metrics change.

The following are lessons learned while implementing and evaluating the PADME algorithm:

- **PADME scales well with high critical section request workloads.** The results of our network simulation in Section 4 showed that the PADME algorithm can reduce synchronization delay to our target (a single message transmission between critical sections). Placing high priority participants close to the logical root and using a Level Priority Model results in low jitter and latency, which is ideal for search and rescue missions presented in Section 2.
- **Distributed computations like the PADME algorithm are harder to implement than centralized solutions**. In our implementation for the network simulator, we combined the message processing and critical section requests into a single event loop, which caused some irregularities that might be hard to trace for application developers (this would not necessarily be a problem for middleware developers – since critical section requests would be generated via the user or application layer). As with all distributed computations and algorithms, care must be taken during testing and development phases.
- **Configuring PADME's four models may be daunting to new users,** who may not understand the complexities of using different Priority, Request, Reply, and Release models together. Our future work is therefore developing model-driven middleware tools [7] that aid in this configuration and ease new developers or users into using the PADME algorithm.
- **The user-provided spanning tree can hinder performance if implemented incorrectly**. PADME will work with any spanning tree (no matter how inefficient), but performance and QoS may suffer. Again, users may be helped by model-driven middleware tools that create an optimized spanning tree from a user-specified scenario.

The PADME algorithm code and tests/simulator used for the results in Section 4 are available for download at our project website [6].

References

1. Agrawal, D., Abaddi, A.E.: An Efficient and Fault-tolerant Solution for Distributed Mutual Exclusion. In: ACM Transactions on Computer Systems, vol. 9.1 1-20 (1991).
2. Apache Software Foundation. Hadoop Project Site. http://hadoop.apache.org/core
3. Argonne National Laboratories. MPICH Project Site. http://www.mcs.anl.gov/research/projects/mpi/mpich1
4. Bertier, M., Arantes, L, Sens, P. Distributed mutual exclusion algorithms for grid applications: A hierarchical approach. In: Journal of Parallel and Distributed Computing, vol 66.1, 128-144 (2006).
5. Cao, J., Zhou, J., Chen, D., Wu, J.: An Efficient Distributed Mutual Exclusion Algorithm Based on Relative Consensus Voting. In: 18th International Parallel and Distributed Processing Symposium, vol. 1, 51-61 (2004).
6. Edmondson, J.: QoS Enabled Mutexes, http://code.google.com/p/qosmutex.
7. Gokhale, A., Balasubramanian, K., Balasubramanian, J., Krishna A., Edwards, G., Deng, G., Turkay, E., Parsons, J., and Schmidt, D.: Model Driven Middleware: A New Paradigm for Deploying and Provisioning Distributed Real-time and Embedded Applications. In: Elsevier Journal of Science of Computer Programming: Special Issue on Foundations and Applications of Model Driven Architecture (MDA), vol. 73.1, (2008).
8. Housni, A., Trehel, M. Distributed mutual exclusion by groups based on token and permission. In: Proceedings of the ACS/IEEE International Conference on Computer Systems and Applications, 26-29 (2001)
9. Kshemkalyani, A.D., Singhal, M.: Distributed Computing: Principles, Algorithms, and Systems, Cambridge University Press, Cambridge, UK. 538-543 (2008).
10. Lamport, L.: Time Clocks and Ordering of Events in Distributed Systems. In: Communications of the ACM, vol. 21.7, 558-565 (1978).
11. Maekawa, M.: An Algorithm for Mutual Exclusion in Decentralized Systems. ACM Transactions on Computer Systems, vol. 3.2, 145-159 (1995).
12. Object Management Group. Data Distribution Service for Real-time Systems. http://www.omg.org/technology/documents/formal/data_distribution.htm
13. Object Management Group. Specialized CORBA Specifications. http://www.omg.org/technology/documents/specialized_corba.htm
14. Renaud, K., Lo, J., Bishop, J., Zyl, P., Worrall, B.: Algon: A Framework for Supporting Comparison of Distributed Algorithm Performance. In: Eleventh Euromicro Conference on Parallel, Distributed and Network-Based Processing, 425-433 (2003).
15. Ricart, G., Agrawala, A.K.: An Optimal Algorithm for Mutual Exclusion in Computer Networks. In: Communications of the ACM, vol. 24.1, 9-17 (1981).
16. Singhal, M.: A Dynamic Information-structure Mutual Exclusion Algorithm for Distributed Systems. In: IEEE Transactions on Parallel and Distributed Systems. vol. 3.1, 121-125 (1992).