Boston College
Computer Science Department

Senior Thesis 2003
Jonathan Pearlin
A Distributed Mutual Exclusion Algorithm Using Multicast Communication
Prof. Robert Signorile

Abstract

The purpose of this project is to integrate a three-dimensional model of a real-world environment into a distributed system based on multicast communication between nodes in the system. The development of three-dimensional environments (3DEs) revolutionized the computer graphics industry, in particular the gaming industry, by bringing computer graphics one step closer to replicating reality. The developed application uses a 3DE of a dormitory (specifically Ignacio Hall on the Boston College campus) to serve as a backdrop for a distributed system in which users compete against each other for resources and objectives (such as health powerups and to obtain a winning score). The development of a distributed mutual exclusion algorithm operating via multicast communication was necessary to ensure the proper performance of the developed distributed system. The created algorithm takes advantage of the characteristics of a multicast network and various synchronization mechanisms, such as a timestamp and the election of a central arbitrator from the multicast group. These synchronization safeguards not only enforce mutual exclusion, but also aid in the enforcement of reliable communication between nodes in the distributed system. The implementation of a dynamic rotating server selected from among the members of the multicast group, in combination with a timestamp applied to all messages sent between nodes in the system, forces the ordering of messages sent within the system. This adds a level of predictability and stability to developed system and its communication mechanism. The final implementation and integration of the 3DE, the distributed mutual exclusion algorithm and the reliable system based on the multicast communication

protocol result in a distributed, three-dimensional world, which mimics the everyday interaction between independent people vying for shared resources.

The presented algorithm in this paper uses a combination of various mutual exclusion techniques in conjunction with synchronization techniques and multicast communication. The goal is to provide an efficient, low traffic algorithm that provides mutual exclusion to a distributed system with shared resources. Previous work in the field suggests that optimization in distributed mutual exclusion algorithms is related to the number of messages required to achieve the mutual exclusion of shared resources. The proposed algorithm promises to achieve this in a constant number of required messages, which is the result of the use of multicast communication. The developed algorithm was applied to the aforementioned distributed system in order to display its functionality and practically in the world of distributed systems.

1. Introduction

Distributed systems utilize a network of computers in order to share the workload of an application evenly amongst the members of the network. Such a system must not only coordinate between processes in the system, but must also provide the same functionality and assurances that one finds in a non-distributed program. These assurances include the mutual exclusion of access to shared resources among processes vying for entry into a critical section. Mutual exclusion consists of the prevention of deadlock between processes and the prevention of the starvation of a process attempting to acquire entry to the critical section. Without these assurances, one cannot rely on a system where processes communicate via messages in order to gain entrance to a critical

section to be safe. Mutual exclusion in a distributed system is therefore required to not only ensure that the work is evenly distributed between processes, but that resources available to the system are also shared evenly and fairly.

Algorithms that provide mutual exclusion to a distributed system are evaluated primarily on the bandwidth utilized to communicate with the other computers in the system, the effect the algorithm has on the throughput of the system, and the effect of synchronizing the computers in the system. The current body of work in this field is primarily focused on enforcing mutual exclusion in a distributed system based on using a point-to-point protocol for means of inter-node communication. Algorithms that meet this condition include message-passing algorithms, election algorithms and token-based algorithms. The goal for all of these algorithms is ultimately the same: the reduction of the number of messages that is required to be exchanged between processes in order to enforce mutual exclusion. By reducing the number of messages needed to ensure mutual exclusion, these algorithms attempt to improve the optimality of the overall system in terms of bandwidth and throughput.

The idea behind the algorithm presented in this paper is to create a distributed mutual exclusion algorithm that provides mutual exclusion using multicast communication between the members of a distributed system. In order to realize this goal, the developed algorithm draws from the various techniques developed to enforce mutual exclusion in a distributed system, such as message passing, the election of a central arbitrator and synchronization [1,2,6]. The proposed algorithm relies on multicast communication (point-to-multipoint) between nodes as the underlying protocol for the transmission of messages in the system. Because the network uses a broadcast protocol

(and not a point-to-point protocol), the number of messages that need to be exchanged to secure entry into the critical section is greatly reduced. This is very different from previously proposed algorithms, which rely on nodes either knowing about all other nodes in the network or at least knowing about neighboring nodes in the network. This choice of a broadcast protocol allows the proposed algorithm to guarantee distributed mutual exclusion with the passing of a constant number of messages between nodes in the network.

The development of the proposed algorithm in this paper took place as a part of the creation of a larger distributed application. The design and implementation of a three-dimensional distributed world in which users compete against each other to obtain resources and objectives required a means of ensuring the mutual exclusion of such shared resources. This type of project presents a unique opportunity to use the characteristics of multicast communication in order to create a distributed mutual exclusion algorithm that not only has the same properties of a point-to-point distributed mutual exclusion algorithm, but also achieves mutual exclusion with the passing of a constant number of messages. Furthermore, the use of multicast communication places greater emphasis on certain features (such as the use of a timestamp and central arbitrator) in the developed algorithm in order to provide synchronization that one would find in a point-to-point communication-based distributed mutual exclusion algorithm. Overall, the developed application presents the framework necessary to create a distributed mutual exclusion algorithm for a multicast communication-based distributed system.

2. Previous Research

Providing mutual exclusion to a distributed system requires the consideration of several important factors, including the way in which processes communicate with each other, the type of network on which the system is based and the coordination of processes in the system. Distributed mutual exclusion algorithms achieve this coordination by using techniques such as message passing or token passing to communicate between nodes [2,5]. Furthermore, these algorithms are often based on point-to-point communication in which each node must communicate directly and separately with any other node in the network it wishes to address. Because of this behavior, the focus of such algorithms has been on the reduction of the number of messages that are required to maintain mutual exclusion while preserving synchronization between the processes in the system.

The problem of synchronization in a distributed system is often addressed by utilizing a logical clock shared between all members of the system. The use of a logical clock in the form of a sequence number or timestamp, as proposed by Lamport, imposes ordering on the events that occur in the system [1]. Lamport determined that there must be some synchronized timing mechanism shared by all the members of a distributed system in order for one to accurately tell if the system is working properly. To be specific, timestamps should be utilized to ensure the ordering of messages sent between the members of the distributed system [1]. After receiving a timestamped message, the receiving node must set its clock to be later than that of the process that sent the message. This is because the event of sending the message occurred before the event of receiving the message. Lamport described this condition as the “clock condition:”

For any events a,b: if a ? b, then C(a) < C(b) [1].

This condition states that if in the ordering of events in the system, event *a* occurred before event *b*, then it can be deduced that the value of the logical clock at the time event *a* occurred is less than the value of the clock at the time at which event *b* occurred. If the clocks satisfy this condition in the distributed system, then the clocks can be used to totally order all events that occur in the system [1]. Lamport concluded that the total ordering of events in a distributed system ensures that all critical section requests and their subsequent relinquishments occur in order [1].

While coordination of the processes in a distributed system is an important consideration in the design of a distributed mutual exclusion algorithm, the method of communication between processes is crucial to the effectiveness of the algorithm. Much of the work in the field of developing distributed mutual exclusion algorithms have been based around the problem of reducing the number of messages necessary to ensure a safe entry into the critical section. Ricart and Agrawala proposed a "message passing" algorithm that requires 2 (N – 1) messages in order to achieve distributed mutual exclusion in a point-to-point communication-based network [2]. In their algorithm, "sequence numbers" (like those proposed in Lamport's paper) are used to create a total ordering of events in the system. A node wishing to enter the critical section sends a request to every other node in the network (this requires N – 1 messages to be sent) and waits for a response from all other nodes (another N-1 messages). The algorithm addresses node failures by using a "timeout-recovery mechanism" to determine if a node is no longer responding to requests [2]. This algorithm is a distributed algorithm, as each node must process the same number of messages for any given event. While assuring

mutual exclusion, the algorithm has a large overhead in terms of the messages needed to achieve mutual exclusion.

Using Ricart and Agrawala as a basis for a distributed mutual exclusion algorithm, other algorithms have sought to improve on the cost of communication between nodes by using different types of messages or methods of communication [3,4,5]. Suzuki and Kasami proposed two different distributed mutual exclusion algorithms – one based on message passing and another based on token passing. In their message-passing algorithm, "grant messages" are used to reduce the number of messages needed to attain mutual exclusion. Nodes only send a "grant message" to a node that has requested entrance to the critical section and not to every other node in the system like in Ricart and Agrawala's algorithm [3]. This sacrifices the size of the message in order to avoid communicating with all other nodes in the system.

The token passing technique (also developed by Suzuki and Kasami) seeks to reduce the number of messages required to ensure mutual exclusion by changing the method by which processes communicate. By passing a "privilege" token to the next node in line to use the critical section, this algorithm presents a distributed mutual exclusion solution that requires at most N messages (where N is the number of nodes in the system) [5]. However, this algorithm is prone to greedy processes (because of its first-come, first serve nature) and "token chasing" (a condition in which a node is constantly following the token from one node to another, but never obtains it).

Like Suzuki and Kasami, Maekawa used the algorithm developed by Ricart and Agrawala to create an algorithm that sends $C \sqrt{N}$ messages to ensure mutual exclusion, where C is a constant between 3 and 5 [4]. This algorithm utilizes a non-weighted voting

(or election) scheme, which uses the intersection of subsets of nodes to find a common node between two competing nodes to act as an arbitrator that grants access to the critical section. This is similar to the election algorithm developed by Garcia-Molina, in which nodes decide on a leader to serve as a central arbitrator [6]. Maekawa's algorithm addresses node failure by changing an active node logically into the failed node. By using a single node to arbitrate requests for the critical section, Maekawa's algorithm is distributed, as each node arbitrates for the same number of nodes [4].

The algorithm proposed in this paper draws from the techniques of all of the aforementioned algorithms. The developed algorithm promises to obtain mutual exclusion in a constant number (one or two) of messages per request for entrance into the critical section: one message to contact the "server" with a request and one message back to entire system informing the group of the server's decision. It can be shown that this algorithm is distributed and that it provides mutual exclusion to the system (the term "distributed" is used to describe the equal number of turns that each node takes as the central arbitrator in the system). The methods used in this algorithm combine the election techniques of Maewaka and Garcia-Molina with the message passing technique of Ricart and Agrawala and Suzuki and Kasami. Furthermore, the timestamp mechanism developed by Lamport is imposed on the system in order to enforce the synchronization of information and the ordering of events. This is crucial to an algorithm based on multicast communication in which order of delivery is not guaranteed.

3. The Proposed Algorithm

The nature of a distributed system built on multicast communication plays a large role in the development of any distributed mutual exclusion algorithm. Not only must such an algorithm assure mutual exclusion, but it must also be mindful of the caveats of multicast communication between members of a network of computers. Members of a multicast group may request resources at any given time, as well as exit and enter the group. Furthermore, because the communication protocol used in the development of the distributed application is the multicast communication protocol (and nodes therefore do not know directly about one another), greater safeguards must be put into place to prevent starvation and deadlock in regards to mutual exclusion. This places more of a burden on the node acting as the arbitrator in the system. Not only must the node acting as the server handle requests at any time, it is also responsible for maintaining the integrity and synchronization of the group. The proposed algorithm for distributed mutual exclusion using multicast communication makes use of timestamps for synchronization and logical identification numbers to reference nodes in the multicast group. The idea behind this is to provide the same security and assurances to a distributed system based on multicast communication that algorithms developed for point-to-point communication protocols provide.

The proposed algorithm is based on a combination of message passing and election based distributed mutual exclusion algorithms. At all times, the algorithm has a node that has been "elected" to serve as the central arbitrator for resource requests for a fixed term (the specifics of this term are discussed in section 3.1). When a node receives an incoming message, its first course of action (whether it is currently the server or not) is

to determine if it is its turn to be the server by examining the timestamp value contained in the message (this is explained in detail in section 3.3). Server status rotates from one node to the next, giving all nodes in the system a turn to act as the server and handle an equally distributed amount of work. Any node wishing to request a resource or enter the group does so by issuing its request to the entire multicast group. The server (who is just another node in the group) listens for such requests and processes and responds to them appropriately. Once the server node has made a decision regarding the resource request, it broadcasts its response to the entire group. The requesting node, upon receipt of this response, enters the critical section if it has been determined by the server that it is safe to do so. All nodes (other than the node that has been granted access to the resource) simply mark the resource as in use and wait for the node with access to the critical section to broadcast a release command before attempting to request the resource again. The algorithm is described in detail below:

1. Node A, wishing to enter the critical section, sends a request to the multicast group requesting the use of resource R.
2. Node B, who is currently acting as the server for the group, receives the request for resource R from Node A. Node B determines that resource R is available and locks resource R for Node A. At the same time Node C issues a request for resource R as well. Before dealing with the next incoming message (which is Node C's request for resource R), Node B sends its response to the group regarding the status of resource R.
3. Node A receives the response to the group and notices that the server has granted Node A access to resource R. Node A enters the critical section and

begins to use resource R. At the same time, Node C also receives the response and sees that resource R is in use by Node A. It marks resource R as in use.

4. Upon finishing its work in the critical section, Node A issues a release message to the entire group, indicating that it is done using the critical section. All nodes, including Node C, mark resource R as being available again. Node C is now free to issue a request for resource R.
5. Steps 1-4 are repeated as nodes wish to access the critical section of the system.

3.1 The Role of the Server

A node that has taken on the role of the server in the system has the duty to fulfill requests for resources in the system. The server's purpose is to provide an extra layer of synchronization in order to assure that all nodes have the same view of the state of the entire system at any give time. The server achieves this synchronization by being the only node with the authority to grant access to a critical section. The time at which a given node becomes the server is relative to the number of nodes in the system and the current value of the timestamp (another layer of synchronization, which is discussed in section 3.3). The timestamp value can only be incremented by the server and is incremented when a resource transaction occurs (the value is incremented before the server sends out its response). As members enter and leave the system, every node remaining in the system recalculates the starting and ending time for its turn as the server.

This is done to ensure that deadlock will not occur by assigning every possible timestamp value to at least one of the remaining nodes in the system.

The proposed algorithm also requires the server to maintain the integrity of the group in order to prevent non-responding nodes from becoming the server and plunging the group into an unstable state. When a node starts its turn as the server (by noticing that the timestamp of the message it just received is equal to its starting time to be the server), the node runs the following code (replicated here in psuedocode):

```
If(I_AM_SERVER && timestamp == start_time)
{
     pingAllMembers( );
     setMyActivity( ACTIVE );
}
```

Figure A: Server module to request membership verification from all other nodes.

The server broadcasts a message to the entire group and waits for a response from each member it believes to still be in the group. It also sets itself to "active" because it is still a member of the group. The server waits to receive replies to the ping until the timestamp is equal to one less than the node's stopping time, as demonstrated by the following code:

```
If(I_AM_SERVER && timestamp == (stop_time – 1))
{
     determineNextServer( );
}
```

Figure B: Server module to determine which node will be the next server for the system.

At this point, the server is almost done with its turn as being the central arbitrator and therefore must decide if the next server is still capable of taking on the role of server. The current server determines which nodes it has received a response from and increments the timestamp by a certain value in order to skip over an "inactive" node. This safeguard ensures that nodes that appear to still be in the group, but have somehow

stopped responding to the group, do not become the server and disrupt the mutual exclusion and synchronization the algorithm provides to the group.

The server's role in the system also includes the arbitration of requests from nodes in the system. The server's identity does not need to be known to the rest of the group in order for a member to communicate with the server. The broadcast nature of multicast communication is one of the advantages of using multicast as the communication protocol in the system. Furthermore, this behavior reduces the number of messages necessary to secure entry into the critical section, as is discussed in detail in section 5 of this paper. A node wishing to access the critical section simply broadcasts its request to the group. Non-server members of the group simply ignore requests for resources, as it is not their place to fulfill such a request. The node that is currently acting as the server is equipped with the following code to deal with incoming messages:

```
received_msg = receive( );

If(timestamp >= start_time && timestamp <= stop_time)
      I_AM_SERVER = true;
else I_AM_SERVER = false;

If(received_msg == membership)
{
      if(received_msg.key() == request_to_join)
      {
            incrementTimestamp( );

            if(open_spot_exists)
            {
                  send( OK_TO_JOIN );
                  calculateMyTurnAsServer( );
            }
            else send( GROUP_FULL );
      }
      else if(received_msg.key() == request_to_leave)
            {
                  removeLeavingMember( );
                  calculateMyTurnAsServer( );
                  if(no_longer_my_turn)
                        I_AM_SERVER = false;
```

```
                }
    }
    else if(received_msg == system_state)
    {
        if(received_msg.key() == request)
        {
            if(resource_available)
            {
                lockForRequestingNode( resource );
                incrementTimestamp( );
                send( RESPONSE );/* to entire group */
            }
        }
        else if(received_msg.key() == release)
        {
                unlockResource ( resource );
        }
        else if(received_msg.key() == ping)
                memberActivity[sender.id] = ACTIVE;
        else {
                /* system state update – i.e.
                   any type of personal information
                   about a node that it needs to
                   notify other members about
                */
            }
    }
```

Figure C: Server module to process incoming messages.

As we can see from the code above, the server has to deal with two different types of requests: a request to join the group and a request for a resource available to the group. In order to maintain the integrity of the group, the server is the only node allowed to welcome nodes into the group (since it is a multicast group, nodes do not know who the members of the group are without some imposed system to keep track of the members). This prevents one node from allowing a node to join the group while another node denies the same node's request to enter the group (basically, it provides another layer of synchronization and strengthens the mutual exclusion on system resources). By centralizing this authority, the algorithm allows the system to remain dynamic in terms of membership without compromising the system's structure or resources.

Like the module to grant membership to new nodes (provided there is room in the system for new members), the module to deal with resource requests is also centralized in the server node. The server must decide, upon receiving a resource request, whether or not the request can be granted. Regardless of its decision, the server broadcasts its response to the entire group (by using the underlying method of multicast communication). This saves the server the message traffic associated with most point-to-point distributed mutual exclusion protocols, which require central arbitrators to send messages to each node in the group individually (as discussed earlier, other algorithms try to reduce the number of messages needed by issuing "grant messages" to only the requesting node [3]). In the proposed algorithm, the server needs to only broadcast to the group the logical identity of the node (using the logical identification numbers given to members as they join the group) that has been granted access to the resource. The algorithm administers resource requests in a first-come, first-server manner and allows the server to request resources. While this may appear to cause starvation, it will be shown that because a server's turn is not infinite, each group member has the same opportunity to enter the critical section.

3.2 The Role of the Non-Server

Although the majority of the pressure is put on the node acting as the server to maintain mutual exclusion for the system, the nodes that are not currently acting as the server still play an important part in preserving the integrity and synchronization of the group. Furthermore, it is important to understand how a node issues a request for a resource and how a node handles the server's decision in order to fully understand how

mutual exclusion is achieved. It is also important to note that the nodes that are not currently acting as the central arbitrator are capable of becoming the server without affecting the mutual exclusion of shared resources. This dualistic nature of the members of the group is what maintains the distributed nature of the algorithm.

The proposed algorithm requires the non-server nodes to follow two guidelines when interacting with other members of the group. Firstly, all nodes must help the server node maintain the integrity of the group by notifying a new node of their existence. This frees the server from having to send the new member information about each member who is currently in the group. This also distributes the work required to allow a new member into the group amongst all the current members of the group. The second guideline that a non-server node must follow is that it must act "passive" in regards to resource requests. That is to say that a node that issues a resource request does not wait for a specific response (it does not block itself waiting for a "yes" from the server). Instead, a requesting node continues executing its workload and accepts whatever decision the server makes. The following is pseudo-code demonstrates the modules of a non-server node:

```
received_msg = receive( );

If(timestamp >= start_time && timestamp <= stop_time)
      I_AM_SERVER = true;
else I_AM_SERVER = false;

if(received_msg == membership)
{
      if(received_msg.key() == request_to_join)
      {
            addNewMember( );
            send( OK_TO_JOIN, myID );
            calculateMyTurnAsServer();
      }
      else if(received_msg.key() == request_to_leave)
            {
```

```
                    removeLeavingMember( );
                    calculateMyTurnAsServer( );
                    if(no_longer_my_turn)
                         I_AM_SERVER = false;
               }
     }
     else if(received_msg == system_state)
     {
          if(received_msg.key() == response)
          {
               processRequestResponse( );
          }
          else if(received_msg.key() == release)
          {
               unlockResource ( resource );
          }
          else if(received_msg.key() == ping)
          {
               send( myID );
          }
     }
     else {
               /* system state update – i.e.
                  any type of personal information
                  about a node that it needs to
                  notify other members about
               */
          }
```

Figure D: Non-server module to process incoming messages.

Similarly to the code for the server node, a non-server node first checks to see if it is its turn to be the server upon reception of an incoming message. As mentioned previously, this code is necessary to keep the distributed nature of the algorithm in tact and to ensure that there is an active server present in the system at all times. The non-server's role in welcoming a new member into the group also plays an essential part in both maintaining the distributed nature of the system and the integrity of the group. When a new member is admitted to the group, the server broadcasts a message to every node (including the new node) that the new player is being allowed into the group. This message contains the number of members currently in the group as well as the new node's logical identification number, which is used to address the node directly. The

non-server nodes receive this message as well and add the new node to their list of the current members of the group. Each node must then send out their information to the new node (since the new player's identification number is now known, this information can be addressed directly to the new node). The new node does not become a member of the group until it has received a response from every member currently in the group. This ensures that every node has the same picture of the composition of the membership of the group. This module also distributes the work for admitting new members to the group by making every node take part in welcoming in a new node regardless of whether their status is that of a server node or non-server node.

While the ability to ensure that new members are admitted to the group correctly is an important feature of a multicast group (where nodes can join and leave the group arbitrarily), a non-server node also has the ability to request system resources by communicating directly with the server ("directly" in the sense that only the server node responds to request messages). Because of the nature of multicast communication (all messages are broadcasted to the group and not sent to only one member), the non-server nodes in the proposed algorithm are "passive" when it comes to resource requests. This is the opposite of most mutual exclusion algorithms, which cause a node to block on an outstanding request for a resource until it is fulfilled. Such behavior would cause some of the distributed features of the algorithm to cease to exist (such as the ability to help the server welcome new members into the group). Instead, a non-server node issues a request and continues on with its workload. When the server broadcasts the response message to the whole group, all nodes take in the response and process it, regardless of whether or not they issued a request. Therefore, in essence, the server dictates to all the

other nodes in the group which node has secured the resource. This is done to maintain an accurate and synchronized picture of the state of the system. As mentioned earlier, there is no queuing involved, as the algorithm is a first-come, first-serve algorithm. When a node is done with the resource, it broadcasts a “release” command to the group. All nodes process this release command, freeing the resource for any other node to now request that resource. The distributed nature of the entire proposed algorithm in this paper ensures that the server node will address all requests for shared resources.

3.3 The Timestamp and Synchronization

The importance of the synchronization of events in a distributed system cannot be over emphasized. As pointed out by Lamport, a distributed system can be determined to be working properly if one can establish the ordering of events in the system [1]. The distributed mutual exclusion algorithms described earlier in this paper all use some form of synchronization to ensure that events are processed in the correct order of occurrence. However, these algorithms deal with fixed networks of computers using point-to-point communication. The use of multicast communication adds an extra level of complexity to the notion of synchronizing events in a distributed system. Here, nodes do not communicate directly with one another and cannot be sure that their message has reached every other member in the group (this is the nature of multicast communication and must be assumed for the purposes of creating synchronization). This means that all nodes in the group cannot be allowed to control a logical clock apparatus (because there is no way for nodes to communicate directly to determine the value of the clock). Instead, there must be a central source that regulates the clock in order to force synchronization upon

all of the nodes in the system. Naturally, the server node was chosen to be in charge of synchronizing the logical clock created for the system (specifically, the algorithm uses a simple timestamp that acts like a real clock – when it equals a prescribed limit, the clock is reset to zero). By designating the server as the only node with authority to change the value of the clock, the algorithm ensures that all clocks in the system will be synchronized and that the workload of the system stays distributed equally.

The management of the synchronization mechanism is crucial to the performance of the proposed algorithm. The current server node increments the timestamp when it receives a message that pertains to a resource or membership request. The following diagram displays this process:

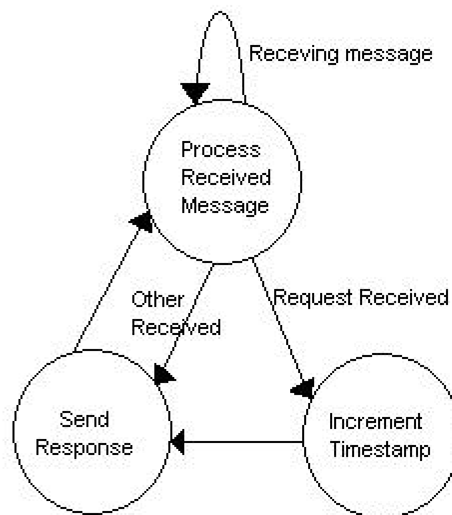

Figure E: State diagram showing when the server node increments the timestamp value.

As we can see from the diagram, the timestamp is increased after this message is received and right before a response is sent out to ensure that subsequent incoming messages are not discarded. The server has the sole responsibility and authority to change the value of the timestamp, which derives from its role as the central arbitrator in the system. This is the reason that synchronization can be achieved even with the use of multicast

communication. By subjugating all other nodes under the authority of the server node, the system is assured that the synchronization mechanism will remain stable.

While the inclusion of a timestamp is necessary to ensure that ordering of events in the system is possible, an issue does arise regarding the rotating server status. The server designation (as discussed earlier) is based on a dynamic calculation, which takes into consideration the number of members currently in the group, the current value of the timestamp, a node's relative position in the group based on its logical identification number and a constant which represents the number of turns each node will serve as the server. For instance, if there are four members in the group, where each member serves five turns as the server, node zero would be the server when the timestamp value falls between zero (node ID * turn value) and four ((node ID * turn value) – 1). Therefore, because a node's status as server is not fixed and changes dynamically based on its relative position in the group (numerical ordering based on identification number), the server node must be consciously aware of the current timestamp and when its turn comes to an end. A potential problem can arise when the server is on its last turn and receives a request which increments the timestamp. For instance, if node two is currently the server on its last turn as server and receives a resource request from node three (who is to be the next server), node two would receive the message, increment the timestamp, and send out the response. Node three, upon receipt of the message, would check the timestamp of the incoming message and update its local timestamp value (as the incoming timestamp value is higher than its current value). As a result of this action, node three now becomes the server, which does not know how to handle a resource response (because a server does not need to request resources externally). In order to deal with this potential

synchronization problem, a “SYNC” message was introduced to the algorithm to force synchronization on a server’s last turn. The following diagram represents this addition to Figure E:

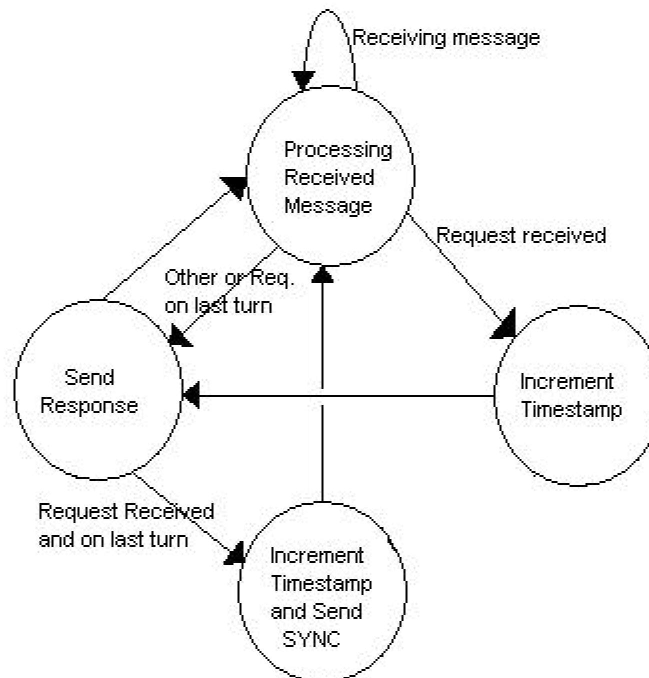

Figure F: State diagram showing the addition of the SYNC message to server module

This gives the server the power to respond to the current resource request and then increment the timestamp value (as is part of the algorithm when a resource request is received) without having to worry about creating the situation described in detail above. If the server is currently on its last turn, it sends its response, then updates the timestamp value (effectively ending its turn as server), and sends out the “SYNC” message with the new timestamp value. All nodes accept the timestamp value contained in a “SYNC” message, regardless of its value. This is because the value is guaranteed to be correct, as the only node that can issue such a message is the acting server. This necessary safeguard is the result of the server apparatus and its implementation. By developing a system based on the dynamic designation of a node to act as the server (a modification on

election-based mutual exclusion algorithms), the addition of a special type of message to ensure synchronization and to prevent a node from taking on the role of the server prematurely is essential. This added safety feature protects the synchronization of the system's logical clock and as a result, protects the distributed nature of the algorithm.

4. Assertions

4.1 Mutual Exclusion

In order for a distributed system to achieve mutual exclusion, no two nodes should have access to the same shared resource at the same time. This means that a node must exit the critical section before any other node can gain access to the same critical section.

Assertion: *The proposed algorithm achieves mutual exclusion.*

Proof: In order to prove that the proposed algorithm achieves mutual exclusion, we must assume the contrary condition: two nodes can both have access to the same shared resource at the same time. In order for two nodes to have access to the same critical section at the same time, one of two cases must be possible:

1. Node A must have already been in the critical section at the time in which Node B entered the same critical section.
2. Node A and B simultaneously entered the critical section.

In the first case, node B is granted access to the critical section that is already occupied by node A. This means that the server node has made two grant responses for the same critical section without receiving a release from node A. This is impossible with the proposed algorithm. Upon receiving a request for the critical section from node

A, the server node would see that the critical section is still available. It would lock the critical section (mark it as in use by node A) and issue a broadcast message to the group stating that node A now has permission to use the critical section. While this is occurring, assume that node B also issues a request for the same resource. While waiting for its response, node B receives the message from the server indicating that node A has been given the critical section. Node B would therefore mark the critical section as being in use by node A. This also means that node B would not be able to enter the critical section until node A broadcasts its release message to the entire group (this is because the server node cannot issue permission to the same resource until it is released). Therefore, the first case cannot occur and mutual exclusion is preserved.

In the second presented case, the server node receives two different requests for the same resource simultaneously. The server node cannot issue two different responses for the same resource. Instead, the server will issue one response indicating which node has been granted access to the critical section. Because of the "passive" nature of the non-server nodes, the node that is not granted access will accept the decision and mark the resource as in use. When the node that is granted access is done with the resource and broadcasts its release message to the group, the node that was denied in its request will again be able to issue a request for the resource. The use of the central arbitrator to determine access to shared resources ensures that no two nodes can enter the critical section by way of simultaneous resource requests. Thus, mutual exclusion is preserved in the proposed algorithm.

4.2 Deadlock

Deadlock occurs in a distributed system when no node can enter the critical section despite requests being issued for entrance. This is usually caused by outstanding requests for the critical section or a failure in the permission granting authority.

Assertion: *Deadlock cannot occur in the proposed algorithm.*

Proof: In order to prove that the proposed algorithm avoids deadlock, we must assume the contrary condition: deadlock can occur among nodes wishing to access a critical section. This would mean that all of the nodes that have made a request for the critical section are waiting indefinitely for a response. In other words, the central arbitrator in the system has failed to respond to a request for a resource. However, this cannot be the case. A node cannot issue a request for a resource that is already in use, as there is no mechanism to block on a resource. Instead, a node checks its list of system resources to see if a particular resource is available. If it is available, then the node is free to issue a request for the resource. If it is not available, the node periodically checks to see if the resource's user has freed the resource before requesting that resource again. Either way, a node receives a broadcasted response from the server to the entire group informing all nodes as to which node has secured the right to use the resource. Therefore, the node cannot be in a state of constantly waiting for a response on a resource (this could occur if the server node only communicated directly with the node that is being granted access to the critical section and left other requesting nodes in the dark about its decision). This also means that a node receives information about the availability of a resource from the server when any node in the system requests a resource. Because of this, a node will not issue a request for a resource it already views as being in use.

On the surface, this “passive” nature, combined with the periodic checking of a resource’s availability, could appear to cause a race condition, in which nodes are constantly contacting the server asynchronously to find out about the status of a resource. This is not what is meant by “periodically” checking to see if the resource is free. Instead, the node checks its own list of the system resources to see if a resource is available. As soon as the node views the resource as free, it may then reapply for the resource. Because all nodes receive a broadcasted response from the node that possesses the resource when the resource is released, the process is synchronized in terms of when all of the nodes view the resource as available for use. This synchronization therefore avoids a race condition. Thus, deadlock is avoided by using the broadcast nature of multicast communication in conjunction with the “passive” state of a node requesting a resource.

It is also possible for deadlock to occur if no node has taken on the role of the system’s central arbitrator. This situation could arise if the node that is to become the server never receives the “SYNC” message sent out by the current server during its last turn. Because the current server sends out the “SYNC” message and then relinquishes its turn as server, neither it nor the next server believe that they are the active server if the message is lost (which is possible with multicast communication). The developed algorithm has been constructed to avoid permanent deadlock caused by such a scenario. Every node checks the timestamp of each incoming message and determines if it is its turn to be the server before processing the message. This means that even though the correct timestamp value never reached the node that is to be the next server, the correct value is present in all of the other nodes that received the message. Since the timestamp

value is included in every packet sent out, those who send out packets with an invalid timestamp (this would be any node that did not receive the "SYNC" message) will have their messages discarded by those nodes that received the "SYNC" message and have the correct timestamp value. This preserves the correct state of the system. Furthermore, the next time any of the nodes (including the former server) that have the correct timestamp value send out a packet, all of the nodes with the wrong timestamp value (including the node that is suppose to be the server) will finally receive the correct timestamp value. This, in effect, would cause the node that is suppose to be the server to notice that it is its turn to be the server. During this period without a server, the system would remain stable because only a server can make system-changing decisions (such as granting access to a critical section or adding members to the group) and increment the timestamp. This ensures that the system stays in the same state while it is attempting to rectify the lost communication. Such synchronization gives the system some margin of error in terms of the rotating server apparatus. While it may be true that a message or two directed to the server may be lost if this scenario occurs, it does not result in permanent deadlock of the system. This is because of the "passive" nature of requesting nodes, which do not block while waiting for a response. Multicast communication is unreliable and therefore extra safeguards must be put into place in the system in order to allow it to recover from lost packets.

Finally, deadlock could result in the system if a node that has been granted the critical section stops responding to the system. This would mean that a resource could be indefinitely held by one of the nodes and therefore would prevent other nodes from every acquiring that resource (this could also cause starvation, as a node would not be able to

get the resource that it needs). However, the node acting as the server in the developed algorithm has the means to deal with this situation. In order to determine if a node is still responding to the group, the server uses the "ping" apparatus discussed in section 3.1. If, at the end of its turn, the server determines that a node that is currently holding a resource is not responding to the group, the server can reclaim the resource for the group by broadcasting the resource's availability to the entire group. This frees the resource and allows any node that wishes to use the resource next to issue a request for that resource. Therefore, the "ping" apparatus serves the dual purpose of preventing the server status from falling to a non-responding node (as discussed in section 3.1), as well as reclaiming resources from failing nodes. Thus, deadlock (and starvation, as discussed above) is avoided.

4.3 Starvation

Starvation occurs in a distributed system when a node is continuously prevented from accessing the critical section while other nodes are allowed to access the critical section. This can occur in systems where unfair weighting is used to process requests for the critical section.

Assertion: *Starvation cannot occur in the proposed algorithm.*

Proof: In order to prove that the proposed algorithm avoids starvation, we must assume the contrary condition: starvation can occur, preventing a node wishing to access a critical section from ever being granted access. This would mean that the node continuously has its request denied by the central arbitrator, possibly as a side effect of network traffic (the node's messages are constantly arriving too late to be granted

access). However, as mentioned earlier in the discussion of the algorithm, the server node has priority to enter the critical section because it is the node that grants access to the critical section. Therefore, a node may not have a chance to enter the critical section while it is not the server, but upon becoming the server, its requests supersede any requests received at the same time (the server node acts just like a regular node when requesting resources – it does not block on an outstanding resource request). This means that any starving node will not starve forever – it will eventually become the access granting authority (due to the rotating server apparatus) with the ability to grant its own access to the critical section, if it is available. Furthermore, as mentioned previously, the server node has the ability to determine if a node that currently holds a resource has stopped responding to messages from the group. If it finds this to be the case, it has the authorization to reclaim the resource for the group. Thus, starvation is prevented in the proposed algorithm.

5. Message Traffic

The proposed algorithm promises to achieve mutual exclusion in a constant number (one or two) of message exchanges. This is the result of the combination of four characteristics of the developed distributed system: the rotating server apparatus, the logical identification numbering of nodes, the "passive" disposition of non-server nodes and the broadcast nature of multicast communication. The deployment of a node to act as the central arbitrator in the system means that all resource requests must be directed to the server node. Therefore, any node wishing to request a resource must only issue one message, asking the server node (whomever it is – the requesting node does not need to

have direct knowledge of the server to have its request heard) for entrance into the critical section. For the server's part, it needs to only send one response to the entire group (as a result of the "passive" non-server nodes and the broadcast protocol), instead of contacting each node individually to notify them of its decision. Each node accepts the incoming response regardless of whether or not it has requested a resource. This whole transaction requires an exchange of two messages to obtain mutual exclusion. If the server wishes to enter the critical section and it is open, then the number of messages required to secure the critical section is reduced to one (the broadcasted message to the entire group informing the group that the resource has been secured). This constant number of messages exchanged is the minimum number of messages required to ensure mutual exclusion in regards to the proposed algorithm. Here, we can fully see the benefit of using multicast communication in order to reduce the number of messages required to provide mutual exclusion to a distributed system. The proposed algorithm takes full advantage of the nature of multicast communication in order to provide mutual exclusion through the exchange of a constant number of messages.

6. Conclusion

This paper presents an algorithm that provides mutual exclusion using multicast communication to a distributed system. The proposed algorithm combines the effectiveness of message passing and election based distributed mutual exclusion algorithms in order to facilitate mutual exclusion [2,3,4,6]. The use of multicast communication as a means of sharing information between members of the distributed system provides certain benefits and obstacles to the implementation of mutual exclusion

in the system. The most obvious benefit is the broadcast nature of communication between members of the group. Any node wishing to communication with any or all nodes in the group simply broadcasts one message to the entire group. This characteristic of multicast communication can be taken advantage of to reduce the number of messages needed to achieve mutual exclusion. However, at the same time, the unreliable nature of multicast communication, the anonymity of members of the group and the constant fluctuations in the makeup of the membership of the group require tighter constraints on the synchronization of the overall state of the system. In order to achieve mutual exclusion in such an environment, an algorithm must assure the system that each node contains the same picture of the state of the system at any given time. This drawback of communication in a multicast group is addressed in the proposed algorithm by using a timestamp mechanism, an elected central arbitrator and the logical identification of nodes in order to force the synchronization of the system state and communication. The result of these safeguards is assured mutual exclusion in a distributed system based on an underlying multicast protocol network.

Besides the use of multicast communication to connect the computers used in the developed distributed system, the combination of the techniques of message passing and the election of a node to act as the server also effect the implementation of the proposed algorithm [2,3,4,6]. The method of message passing provides a simple and effective way for independent members of a network to communicate with each other. Such a scheme allows all relevant system information to be shared by utilizing the underlying network protocol. Because of the broadcast nature of communication in a multicast group (as discussed above), the use of message passing is essential to bringing to fruition the idea

of an algorithm that assures mutual exclusion in a distributed system built on a multicast network. The drawbacks of using message passing include the need to synchronize messages to ensure order and determine the type of messages that need to be synchronized. The problem of message synchronization is directly related to the problems of synchronization in a multicast group, as discussed earlier. In order to prevent false or outdated views of the system from propagating throughout the group, the system utilizes a synchronization mechanism in the form of a timestamp. This timestamp must be placed in any message that has the potential to update or change another node's view of the state of the system. This control of the validity of the content of a passed message is essential to maintaining the integrity of the system and its state. By imposing order on the technique of message passing through the use of a timestamp and logical clock, the proposed algorithm is able to avoid the potential drawbacks of multicast communication.

Like the inclusion of the message passing technique for communication between nodes in a distributed system, the use of the election of a node to serve as the central arbitrator in the system also effects the implementation of the proposed algorithm. The election of a member of the network to serve as the central arbitrator for the system provides another level of synchronization for the proposed algorithm. By forcing all requests for a system resource to go through a single node, the proposed algorithm ensures that there can be only one decision made (and therefore no conflicting opinions) about the status of that resource. Furthermore, the election of a node to act as the server in the proposed algorithm is used to maintain the distributed nature of the system. However, the drawback of the implementation of some election algorithms is that the

group is not necessarily guaranteed that all nodes will have a turn as the server before repeating nodes. The proposed algorithm addresses this potential problem by adding logical identification numbers to each node, effectively ordering the nodes for their turn as the server. Therefore, each node in the system gets the chance to act as the server and handle the same workload before any node gets a second turn. This is essential to maintaining the equal distribution of work in the system. The inclusion and implementation of the technique of electing a central arbitrator further strengthens the synchronization of the entire system, thus reinforcing the guarantee of mutual exclusion made by the proposed algorithm.

While the proposed algorithm is capable of achieving mutual exclusion in a distributed system built on multicast communication, it can benefit from modifications. As mentioned previously, one of the major problems of a multicast group is that nodes can never be totally sure about the membership of the group. Nodes do not communicate directly and therefore do not have direct knowledge of other members. This is the reason why logical identification numbers (based on the order in which new members join the group) were introduced. Future modifications should focus around stronger fault detection to ensure that the system can avoid the unthinkable – the death of the node that is currently the server. Such a safeguard would temporarily sacrifice the synchronized picture of the system's state in order to name a new server by waiting a certain period of time before relieving the current server of its duties. One approach to implementing this protection would be to use a "bully" algorithm in which a node that notices the failure immediately makes itself the server (this would be a combination of the timeout mechanism of Ricart and Agrawala with Maewaka's node failure detection mechanism)

[2,4]. Drawbacks of such an apparatus would include the possibility of having two nodes acting as the server concurrently. Such a scenario could result due to the relativity by which nodes view the state of the system. What appears to be inactivity on the server's behalf to one node may not appear to be the same to another node. For instance, if the physical connection between node A and the server is cut, node B could still communicate with the server and node A. Node A could label the server as inactive, while node B still views the server as active. This would cause great instability in the system. This would need to be addressed by having all nodes that have taken on the role of the server to decide on a new server as a group (probably by choosing the node with the lowest logical identification number as the replacement server). Therefore, future work should be dedicated to finding an optimal and safe solution to preventing the failure of the currently elected server.

The proposed algorithm provides distributed mutual exclusion for a group of computers that exchange messages using multicast communication. The algorithm allows for nodes to join and exit the group, without compromising the promise of mutual exclusion. It is distributed in the sense that each node takes a turn as the central arbitrator for the system, handling the same number of resource requests before the end of its term as server. The number of messages required to secure entry into the critical section is constant and optimal in terms of the number of messages required to ensure mutual exclusion in the described multicast environment. The algorithm makes use of several well-known distributed mutual exclusion techniques: message passing and the election of a node to act as the server. Therefore, the proposed algorithm is suitable for the

imposition of mutual exclusion on a distributed system based on multicast communication.

References

1. Lamport, Leslie. "Time, Clocks, and the Ordering of Events in a Distributed System." *Communications of the ACM.* Volume 21, Number 7. July 1978. 558-565.

2. Ricart, Glenn and Agrawala, Ashok K. "An Optimal Algorithm for Mutual Exclusion in Computer Networks." *Communications of the ACM.* Volume 24, Number 1. January 1981. 9-17.

3. Suzuki, Ichiro and Kasami, Tadao. "An Optimality Theory for Mutual Exclusion Algorithms in Computer Networks." *Proceedings of The 3rd International Conference on Distributed Computing Systems*. October 18-22, 1982. 365-370.

4. Maekawa, Mamoru. "A vN Algorithm for Mutual Exclusion in Decentralized Systems." *ACM Transactions on Computer Systems*. Volume 3, Number 2. May 1985. 145-159.

5. Suzuki, Ichiro and Kasami, Tadao. "A Distributed Mutual Exclusion Algorithm." *ACM Transactions on Computer Systems*. Volume 3, Number 4. November 1985. 344-349.

6. Garcia-Molina, Hector. "Elections in Distributed Computer Systems." *IEEE Transactions on Computers*. Volume C-31, Number 1. 1982. 48-59.