Performance of Token-based Distributed Mutual Exclusion Algorithms

Scott J. McCallen, Department of Computer Science, Kent State University

Abstract—Distributed systems often use distributed mutual exclusion algorithms in order to ensure only one process enters a critical section at a given time. Here we look at two well-established token-based distributed mutual exclusion algorithms, Raymond's and Suzuki-Kasami’s, and identify and verify their advantages and disadvantages through message complexity and synchronization delay. Additionally, we consider two new aspects of the algorithms which, for Raymond’s, lead to better performance, and for Suzuki-Kasami, expose an inherent issue with token forwarding.

Index Terms—Distributed algorithms, Multisensor systems

I. INTRODUCTION

In distributed systems, there is often a need to ensure that particular portions of code are accessed by only one process at a time. In particular, this is the problem of distributed mutual exclusion which has been studied for many years by computer scientists. Those studies have resulted in several possible solutions, some being permission based, such as Lamport’s and Maekawa’s, and others being token-based such as Raymond's and Suzuki-Kasami's. Token based distributed mutual exclusion algorithms work by passing a single "token" through the system and when a process receives the token, it can enter into critical section, in general. The problem of distributed mutual exclusion is different from the normal concept of mutual exclusion because there are 2 or more independently running processes which have no means of communication other than passing messages over message channels. So unlike mutual exclusion on a single system which can be solved using traditional approaches such as locks and semaphores, distributed algorithms must exchange messages to communicate. Token based distributed mutual exclusion algorithms normally require the exchange of two types of messages: request and token messages. Requests are sent from processes that want to enter a critical section (or on behalf of that process) to a process that holds the token. Similarly, the token is sent to a process that has requested the critical section when the holder has completed its own critical section.

A. Performance

Performance is very important in distributed mutual exclusion algorithms. If an algorithm is too slow or easily becomes deadlocked, perhaps it may be an interesting study topic; it is not a worthwhile solution for real-world applications. In particular, we can measure two specific attributes, those being message complexity and synchronization delay. Message complexity is defined as the number of messages that are sent in a system per critical section entry request and the synchronization delay is defined as the number of messages sent between the release of the token and the next critical section entry multiplied by the average message propagation delay. With these two attributes we can effectively compare the algorithms based on the average number of messages sent per critical section request and the time delay between each critical section entry.

II. EXPERIMENTS

A. The Algorithm and Attributes

For the purpose of this study, I plan to investigate the two aforementioned token-based distributed mutual exclusion algorithms, Raymond and Suzuki-Kasami. I will first conduct experiments to verify their previously found performance measures and then conduct two additional tests (on each algorithm independently) to discover a new attribute of each. Let us consider each algorithm in more detail. Raymond's algorithm works by maintaining a pointer at each node, which is represented by the "holder" variable that points to a node that either holds the token or a node between the current process and the token. In this manner, a tree is maintained in which the root of the tree holds the token. Each process also contains a request queue. When a process wants to enter a critical section, it first adds itself to its own request queue and forwards a request message to its "holder". Upon receiving that request the holder process adds the request to its queue and forwards the request to the next holder until reaching the token. Once the requests have been forwarded to the token holder and the holder wants to release the token, it sends the token to the first request on the queue, and, if there are additional requests on the queue, also sends a request to the
new holder. With this structure, we can intuitively see that requesting and releasing the token (message complexity and synchronization delay) tends to make message sequences traverse the branches of a tree, which is a commonly studied problem. Traversing a tree can be difficult in the worst case (each parent node has one child) or easy in the best case (there is exactly one parent of which all other nodes are children). This becomes a key point in my study. Suzuki-Kasami algorithm is much different, however. When a process wants to enter a critical section, it increases a local sequence number and broadcasts that request and sequence number to all other processes in the system. When another process receives that request and the sequence number is higher than that which had been previously received for the requester, it records that it has received that higher sequence number from that specific process. Upon releasing the token, the token holder checks to see which processes have pending requests (a higher sequence number has been received) and add those processes to the request queue in the token (if it is not there already) and sends the token to the top request queue. Again we can intuitively reason that the larger a system becomes, the more messages that must be sent to request the token (and hence enter a critical section). There can be, however, forwards of the token that take place before a requesting process is allowed to enter critical section, which will be discussed later in this paper.

B. Initial Setup and Tests

In general, the performance of token-based algorithms depend mostly on the number of processes in the system and the number of processes simultaneously contending for the token. For the purpose of these experiments, I assert that the network will be tested under low-load conditions and high-load conditions. Here, low-load is defined as any given state for all computations contains one or less pending requests for the token and high-load is defined as any given suffix for all computations contains requests from N-1 processes in the system before the token holder releases the token. As for the network size, if there are only two processes, a high performance algorithm can be created. However, as the number of processes increases, the synchronization delay and message complexity can be affected, depending on the algorithm in question. I plan to vary the number of processes in the system in order to show significant differences in the two attributes mentioned above. For Raymond's algorithm, I will vary the number of nodes between five (5) and sixty (60) incrementing on an interval of five (5). Similarly, for Suzuki-Kasami, I will vary the number of nodes between five (5) and thirty (30) incrementing on an interval of five (5).

For these experiments I will use TOSSIM, along with Tython, to create simulated scenarios in which to run the processes. Tython, by using control messages, will initiate the initial tree structure and token holder for Raymond's algorithm and will then insert control messages to predefined processes, telling the processes when to request the token and when to release the token. The initial tree structure will be a perfect binary tree, that is, node 0 will be the parent of nodes 1 and 2, node 1 will be the parent of 3 and 4 and so on. Similarly, using Tython, control messages will be inserted to initiate Suzuki & Kasami's algorithm so that one process initially holds the token and additional control messages will specify when a process will broadcast requests for the token and when the holder will release the token.

In order to obtain accurate results, for each system size the simulation will cause a predetermined number of token requests and releases, in that order, by a random node and the token holder respectively. By choosing a random node to make the request, it ensures that there is no predefined order in which the requests are made with respect to the system size or network topology. Each trial will run at least 30 requests and release sequences to ensure that an accurate result will be calculated for each attribute of each algorithm.

III. INITIAL VERIFICATION RESULTS

The purpose of the initial experiments is to ensure a correct implementation of the two algorithms and to begin the experimentation phase. Each result set provided posed questions about the exact calculation of each algorithm's performance which will be discussed.

A. Raymond's Algorithm

Raymond's algorithm produced satisfactory verification results which are, as stated previously, similar to what we would expect for traversing a tree. In these experiments, each node had no more than two children; hence we are considering a binary tree. For the low-load situations, the expected average synchronization delay was \( \log(N)/2 \) which is the average distance between two nodes of a random tree with size N. In my results for the binary tree, I experienced a more specific situation where the average distance between two nodes is \( \log(N)/\log(2) \) (i.e. \( \log \) base 2). My results were consistent with this expected distance. Additionally, the same
3 results were found for the message complexity. For low load situations, the message complexity is twice the synchronization delay (or the synchronization delay is half of the message complexity) because the requests must first travel to the token holder (which is \( \log(N)/\log(2) \)) must travel back to the requester. Hence the total messages exchanged for this exchange is twice the synchronization delay or \( 2[\log(N)/\log(2)] \). The results were much different for high load situations with Raymond’s algorithm. In this situation, all of the nodes except one (the holder) requested the token and then the token was released that many times, hence satisfying all the requests sequentially. The order in which the requests are satisfied depends upon the order in which they are received and entered into the request queue at each node. It is possible that even though a token is passed to a node that has requested the queue, that node will continue passing the token onto another node because its own request was not at the top of its queue. The results of these experiments show that in a high contention system, each critical section request has an average message complexity of four (4) and synchronization delay of two (2). This is due to the fact that, on average, there will generally be at least one node between the requester and the token before the requester is satisfied. Therefore there are two edges to traverse to satisfy the request, making the total number of message sent four (4) and the synchronization delay two (2).

B. Suzuki-Kasami’s Algorithm

Results from the Suzuki-Kasami experiment are straightforward. Because each request requires a broadcast of messages to all other nodes and the token is then sent directly to a requesting process, one would expect that the message complexity is exactly \( N \) for a system of \( N \) processes. The former is exactly true, and, as shown in the test results, the message complexity of a low-contention and high-contention system have a one-to-one correlation with the number of nodes in the system. The synchronization delay is similarly straight forward. Because the Suzuki-Kasami algorithm requires a fully connected system, the token is able to be directly sent to the next request that needs to be satisfied. Therefore, the synchronization delay is exactly one (1) in both a low-contention and high-contention system, which is shown in the experiment results.

IV. PERFORMANCE EXPERIMENTS

While producing the results for the previous experiments, it became clear that there are a few aspects of each algorithm which have not been properly formalized. For the initial tests of Raymond’s algorithm, the tree structure was produced by each parent node containing at most two children. As shown, this leads to a direct correlation in the synchronization delay and message complexity. If this is the case, then it seems reasonable that there would be a correlation between the number of children per node and the same attributes. Realizing this, experiments were setup with Raymond’s algorithm in a low-contention system in which number of children per node would vary. For this experiment, it would not be advantageous to vary the number of nodes in the system because the focus is on the number of children per node so the system size of 50 nodes was established and stays constant. To get a wide range of results, the child count (number of children per node) was varied from one (1) (i.e. a chain) to nine (9). The system of 50 processes was tested at each child count with 40 request and release sequences and...
the results of this experiment are discussed in the next section. While testing Suzuki-Kasami, another phenomenon occurred. Inspecting the algorithm produced more insight into the fact that upon releasing a token, a node is often given higher precedence because of its node identification. If this is the case, then in a high contention system with many processes, a node may have an extremely long waiting time before being allowed to enter critical section. Given this, it became interesting to study the number of token forwards per request in a high-contention. Tests were run with Suzuki-Kasami’s algorithm with the same initial high-contention system setup except a new count array was added to the token in which the contents of the array was incremented each time the token was forwarded to a requesting process. When a requester received the token, it added the token forward count to the message complexity and recorded it as total messages. The results are discussed next.

A. Results

Both experiments produced interesting results. For Raymond’s algorithm, changing the number children per node had a great impact on the message complexity and synchronization delay in a low-contention system. With the system size constant at fifty nodes, the synchronization delay dropped steadily with an increasing number of children per node. With one child per node (i.e. a chain of processes), the synchronization delay was approximately 16 messages on average, whereas a system with 6 children per node had a synchronization delay of only 4 messages on average. Certainly a chain of processes is a rather extreme case, but a system with 6 children per node still has a significant advantage over a system with 2 children per node (the original binary tree case) which had a synchronization delay of 6 messages on average. Increasing the number of children beyond 6 per node showed a slow down in the decrease rate for improvement of synchronization delay and would, as it seems, not lead to significantly better performance. This result has a direct correlation with the diameter of the tree, the smaller the diameter the lower the synchronization delay. With that said, this experiment can be summarized by realizing that, on average, the synchronization delay of a low-contention system using Raymond algorithm will be proportional to log(N)/log(C) where N is the number of processes in the system and C is the number of children per parent node.

The next experiment, which was performed using a varying system size, high contention and concerned Suzuki-Kasami’s algorithm, resulted in a formula that specifies the larger a system is, the longer a node with have to wait for the token. Because Suzuki-Kasami’s algorithm relies on a node’s identification to know whether or not that node has requested a critical section, there is an inherent order which the nodes must be placed in. This order then defines how the requesting nodes will be added to the token’s request queue and therefore how long it will take for the token to reach the last requesting node. In a high contention system where all processes initially make a request for the token and then all those requests are satisfied, the node with the highest identification number (relatively speaking) will have to wait the longest because the token will first be forwarded to all the other nodes. Therefore in a high contention system, the message complexity and token forwards that a node must endure on average, is equal to [(N*N) + sum(1:N)]/N for a system of N processes which are all requesting the token simultaneously. While this correlation is still a linear relationship with the system size, the degree (slope) of the relationship is greater than that of the original message complexity. This result is shown in the experimental result data.

V. CONCLUSIONS

In conclusion, the initial test results verified that in an experimental application, the message complexity and synchronization delay of both Raymond’s and Suzuki-Kasami’s algorithm are as previously found. These results
have been verified by the experimental setup which was created using Tiny OS, TOSSIM and Tython. Additionally, there are added attributes which can be focused on, such as the number of token forwards for Suzuki-Kasami or the advantages of higher child counts for Raymond's. There are, of course, certain advantages and disadvantages to be found with each algorithm. In terms of message complexity, Raymond's algorithm has definite advantages over Suzuki-Kasami. The latter has a linear message complexity that increases with the size of the system, but Raymond's suffers very little when the system size becomes very large, especially in situations with many children per parent node. Suzuki's Kasami can have benefits too, considering it's extremely low synchronization delay for high or low-contention systems (though Raymond's synchronization delay is very similarly in high-contention). Depending on the particular application and the inherent needs of the processes that will be making the critical section requests, the advantages and disadvantages of each algorithm should determine the proper one to implement.

VI. FUTURE WORK

Future plans for this project are to test the two new aspects of the algorithms (child count and average token forwarding) in both low and high-contention systems. In these experiments, high-contention was asserted to be \(N - 1\) processes make requests and then all requests be satisfied. Testing in a continuous high-contention system may also provide interesting results. For the child count, creating a random tree network where there are a random number of children per parent node could produce results similar to the original proposal of Raymond's message complexity and synchronization delay. Additionally, it may be worthwhile to run these tests outside of the simulated environment and load these applications onto the micas in order to test their use in a live system. It is possible that certain tree-based algorithms may perform very well with Raymond's algorithm while other consensus algorithms would work best with Suzuki-Kasami's algorithm. Creating Tiny OS modules and components out of these algorithms for use in the network sensor and Tiny OS community would also be advantageous in order to share these algorithms (both the benefits and pitfalls) in a unified form with the community. This would allow real-world systems to be ran using these algorithms and perhaps users could provide data that would show real-world message complexity and synchronization delay results.

VII. IMPLEMENTATION

Implementation of these algorithms, and other projects throughout the semester, has been anything but trivial. The concept of Tiny OS and its accompanying components (such as TOSSIM and Tython) are relatively simple and straight forward, given the nature of the problem and the tools to solve those problems. Learning and developing with these tools, however, has not been straight forward. The tutorials are helpful to and extent, but I often found the knowledge base (namely the mailing list archives) to be uninformative and outdated. Granted I was able to find answers to most of my questions, I believe there is a lack in user support of Tiny OS and understanding of this system is not readily accessible to a user outside of the given research community.

A. Issues

First and foremost, there were several issues with the message system. While the tutorials explained how to use the message system, they were solutions to the problems that were not yet known. Most of the issues stemmed from two specific system occurrences that are not well documented: message loss and message size. The Suzuki-Kasami algorithm requires that a request queue be contained within the token that is sent between processes. The request queue takes up a linear amount of space as the system size grows, but the message size in TOSSIM was limited to 100 KB for this project. This is why the total number of processes in the system for Suzuki-Kasami was limited to 30 nodes: there wasn’t space available on the token for any more data. This issue and its resolution (increasing the allowed message size in the make file) is not well recorded in the TinyOS documentation other than the fact that the initial message size limit is 29 KB. The other problem, message loss, is even less documented. In the radio models provided with the TinyOS installation, each message that is sent to GenericComm to be transmitted is added to the transmit queue and GenericComm attempts to send it. If the radio is busy (either another messages is being transmitted or received) the other message being transmitted may never be transmitted. This can cause many problems in a simple simulation unless proper precautions are taken to ensure that all messages are properly sent and received. Often this would come in the form of a custom message queue which holds the messages until they can be properly sent and received through GenericComm. This message queue, however, is not provided as part of TinyOS.

There were several issues encountered during implementation using the provided skeletons, though none were unfixable. Most dealt with minor issues which were hard to locate. This leads to another well-known issue with the TinyOS environment, lack of advanced debugging options. Certainly printing the state of each node at certain intervals is a step in the right direction, however the TinyOS environment is far behind most modern debugging options such as direct runtime debugging of source code, breakpoints, variable watch statistics, etc. This becomes an even larger issue when the software is deployed to the motes when little, if any, debugging information is available at all.
In general, using TinyOS, TOSSIM, and Tython has been a learning experience. As in any new endeavor, there is often an initial learning curve. The event driven architecture truly lends itself to distributed algorithms and is related to the standard object oriented programming paradigm. With that in mind, the lack of documentation for the environment often makes it terribly difficult for a beginner to easily begin writing software for these systems. Additionally, once the basics are learned, it can become even more difficult to write complex software as there are small details that can lead to complex problems. More documentation and sample projects that explain why a particular approach was taken could be very helpful.

B. Successes

In general, the implementation of the algorithms was successful, though improvements can be made. A message queue, similar to the one use for Project 2 could be implemented to ensure message delivery. Additionally, the use of Tython for the project was very helpful. Using Tython, I was able to automate most of the experiments and create output files for later access and analysis. The advantages of Tython became very clear and future projects will definitely include room for Tython control messages.

REFERENCES