Fair K Mutual Exclusion Algorithm for Peer To Peer Systems [*]

Vijay Anand Reddy, Prateek Mittal, and Indranil Gupta
University of Illinois, Urbana Champaign, USA
{vkortho2, mittal2, indy}@uiuc.edu

Abstract

k-mutual exclusion is an important problem for resource-intensive peer-to-peer applications ranging from aggregation to file downloads. In order to be practically useful, k-mutual exclusion algorithms not only need to be safe and live, but they also need to be fair across hosts. We propose a new solution to the k-mutual exclusion problem that provides a notion of time-based fairness. Specifically, our algorithm attempts to minimize the spread of access time for the critical resource. While a client's access time is the time between it requesting and accessing the resource, the spread is defined as a system-wide metric that measures some notion of the variance of access times across a homogeneous host population, e.g., difference between max and mean. We analytically prove the correctness of our algorithm, and evaluate its fairness experimentally using simulations. Our evaluation under two settings - a LAN setting and a WAN based on the King latency data set - shows even with 100 hosts accessing one resource, the spread of access time is within 15 seconds.

1. Introduction

In peer to peer (p2p) systems, a distributed k mutual exclusion primitive provides a mechanism to share resources like bandwidth in a decentralized fashion. Consider a central statistics collection server in a p2p system (for example Planet-Lab [13]) that provides some distributed services. The limited bandwidth at the server is the bottleneck here, and in such a scenario, a k mutual exclusion primitive can be used to cap the number of simultaneous connections to the server at k, while providing access to the server in a decentralized manner. We note that while there is a lot of emerging work on distributed data collection and data aggregation [22, 23], the current infrastructure for statistics collection is largely centralized [17]. Another example where k mutual exclusion in p2p systems would be useful is when multiple clients are simultaneously downloading a large multimedia file, e.g., [3, 12]. Finally applications where a service has limited computational resources will also benefit from a k mutual exclusion primitive, preventing scenarios where all the nodes of the p2p system simultaneously overwhelm the server.

The k mutual exclusion problem involves a group of processes, each of which intermittently requires access to an identical resource called the critical section (CS). A k mutual exclusion algorithm must satisfy both safety and liveness. Safety means that at most k processes, $1 \leq k \leq n$, may be in the CS at any given time. Liveness means that every request for critical section access is satisfied in finite time.

For the applications described above, *fairness* of the k mutual exclusion primitive is important, to ensure predictable staleness. In other words, the primitive should ensure timeliness of data (statistics) collection across nodes. Previously proposed algorithms for k mutual exclusion focus on optimizing the mean time to obtain access to the critical section, rarely accounting for metrics like fairness. Thus, new solutions are needed that take fairness into consideration. Our notion of the fairness metric is as follows: amongst a group of homogeneous nodes having a constant critical section execution time, the access time spread for the critical section should be small.

We do not consider fairness to be a binary property; the smaller the access time spread for the critical section, the more fair the mutual exclusion algorithm. There are other ways to measure fairness, e.g., [8] defines fairness as FIFO with respect to request timestamps. Our notion of fairness is more practical, yet it provides FIFO ordering as a side affect.

In this paper, we propose a practical fair algorithm for k mutual exclusion in p2p systems that minimizes the difference between maximum and mean time to access the critical section. We prove that our algorithm satisfies both the safety and liveness requirements. We show that our algorithm provides an order of magnitude better fairness characteristics as compared to the best-known k mutual exclusion algorithm [2], while the asymptotic bounds on both the

[*]This work was supported in part by NSF CAREER grant CNS-0448246 and in part by NSF ITR CMS-0427089.

mean time to access the critical section, as well as the number of messages per critical section entry, remain the same. We also propose a fault tolerant methodology for our algorithm and show that it is resilient against churn.

2. Related Work

Classical mutual exclusion algorithms can be classified into two groups: permission (quorum)-based and token-based. Permission-based algorithms [6,9,16] require one or more successive rounds of message exchanges among the nodes to obtain the permission to execute the CS. In token-based algorithms [4,11,15] a unique token is shared among the nodes. A node is allowed to enter its critical section only if it possesses the token.

The generalization to k mutual exclusion was proposed by Raymond [14], but the algorithm is permission based. Srimani and Reddy [19] proposed a token based algorithm for the k mutual exclusion problem by extending the ideas of Suzuki and Kasami [21]. Their algorithm involves flooding of requests, which makes it poorly suitable to p2p systems. Ricart and Agrawala [16] used a single token with a counter to keep track of how many processes are currently in the critical section. It is shown in [2] that under high load, this system behaves like a single token system. Also, it requires all nodes to communicate in certain stages of the algorithm, making it message inefficient for p2p environments. Also, both [19] and [16] are not fault tolerant and do not consider fairness.

Bulgannawar and Vaidya [2] presented a token based algorithm that uses k separate tokens in the system. Their algorithm uses k instantiations of a dynamic forest tree structure proposed in [11]. The algorithm uses heuristics for choosing a tree among k, to send the request. This decision makes the algorithm unfair, since there is no load balancing of requests. Moreover, the paper does not consider fault tolerance.

In recent years, p2p overlays like Chord [20] have been proposed. It provides flexible route selection and static resilience. Lin *et al* [6] and Moosa *et al* [10] proposed quorum based protocols for achieving mutual exclusion in dynamic p2p systems. The Sigma protocol [6] uses logical replicas and quorum consensus to deal with the system dynamism whereas [10] proposed protocols which combine both token and quorum based approaches.

3. A Fair K Mutual Exclusion Algorithm

3.1. System Model and Problem Definition

We assume that each node has a unique id and can send messages to any other node in the system. The communication channel is reliable and does not duplicate messages. Message delivery to the destination is time-bounded. There are k tokens numbered 1 through k in the system and a node can be in the critical section only if it is holding a token. A node holds a token for only a finite time. A node does not initiate a new critical section until it has exited all previous ones. We initially assume no joins or failures; we later handle failures by relying on a distributed hash table (DHT) called Chord.

The k-mutual exclusion problem involves a group of nodes, each of which intermittently requires access to an identical resource or piece of code called the critical section (CS). At most k nodes, $1 \leq k \leq n$, may be in the CS at any given time. we next define safety, liveness and fairness:

Safety means that at most k nodes will execute the critical section at a time. As the algorithm is token based, only nodes with a token can enter the critical section. In order to provide safety, it is thus sufficient to show that there are at most k tokens in the system at any time.

Liveness means that every request for critical section access will be satisfied within a finite time. This definition requires that neither deadlock nor starvation should occur. Deadlock means that, while there are less than k nodes executing critical section, no requesting node can ever enter the critical section. Starvation occurs when one node must wait indefinitely to enter the critical section even though other nodes are entering and leaving the critical section.

Fairness means that amongst a group of homogeneous nodes having a constant critical section execution time, the access time spread to access the critical section should be small.

3.2. Algorithm Description

Our approach to provide a fair k mutual exclusion can be understood intuitively in a real world scenario. Imagine a cinema hall with k ticket counters. A person entering the cinema hall picks one of the k counters randomly with out the prior knowledge of length of queues at each counter. We can see that, this randomized strategy as in [2] gives a good average time to get the ticket. However, in the worst case if every person enters the same queue i.e., if there is no load balancing among the ticket counters, the maximum time to get the ticket will be very high.

A solution for this real world problem could be to place a moderator (coordinator) at the cinema hall entry who has the information of all the ticket counters. The coordinator could then guide people who enter the cinema hall to ticket counters in a round robin fashion, i.e., coordinator does the job of load balancing the ticket requests among the k ticket counters. Since this approach is centralized (presence of coordinator), we cannot directly adopt this approach for our algorithm.

Consider the following alternative approach: there is a coordinator who has the information of the ticket counters (locations and the counter_number). When the first customer arrives at the entry of cinema hall, the coordinator passes his information about the ticket counters to the customer. Now the customer selects the ticket counter with index counter_number and also acts as the new coordinator. The new coordinator passes the updated information (increment counter_number in a round robin fashion) to the next customer at the cinema hall entry queue. This way the information required to load balance among ticket counters is passed along the customer queue without the need for the centralized coordinator.

The above scenario directly maps to our k mutual exclusion algorithm. We use a dynamic logical tree as the representation of the single queue out side the cinema hall (also called the *privileged* queue) and k distributed token queues as the queues at k ticket counters. The dynamic tree is based on the path reversal technique proposed in [11] for solving single mutual exclusion and uses an expectation of $O(\log(N))$ messages to access the CS. The entry of customers into the cinema hall (becoming coordinators) can be viewed as a single mutual exclusion problem, similarly our algorithm has the notion of a coordinator node. The coordinator node has information about the tails of the k distributed token queues and the counter_number which is to be incremented in round robin fashion while passing the information to the next node in the privileged queue. Every requesting node has to become a coordinator node in order to get assigned to one of k distributed token queues (defined by child pointers).

Now, we give an overview of the single mutual exclusion proposed in [11]. The basic concept underlying the algorithm is path reversal. Path reversal at a node i is performed during transit of node i's request toward the root of the tree structure. As the request is forwarded towards the root, node i becomes the parent of each node on the forwarding path. Also, node i becomes the new root of the forest tree. Thus, the shape of the tree structure, relative positions of the nodes in the tree structure, and the connections change. Finally, all the request for critical section form a single *privileged distributed queue* which is defined by the $next$ pointer and the end of the queue is the $root$ of the remaining dynamic tree.

The round robin entry of nodes ensures that all the k tokens are used equally and in a fair manner. This means that the difference between average and the worst case service times of all k distributed token queues will be small. [2] provides FIFO ordering on requests of individual queues, but fails to provide FIFO ordering on requests of all queues, whereas our algorithm provides FIFO ordering on requests of all queues beyond the coordinator, i.e, all the requests are satisfied in order of their coordinator timestamps. Since we are using an additional constant number of messages compared to [2], average message complexity of our algorithm remains $O(\log(N))$.

We now present our algorithm pseudo code. The local variables that are used to maintain state information are depicted in Table 1. We make use of the following messages in our algorithm.

- **Message_request**(k): request sent by a node k to its $parent$
- **Message_token**: transmission of a token
- **Message_child**(r): assignment of a child
- **Message_token_locations**(a,b): transmission of $token\_locations$:= a , $counter\_number$:= b

```
Initialization
$parent := 0$
$requesting\_cs, in\_child\_queue, is\_coordinator := false$
if node $id < k$ then
    $token\_present, in\_child\_queue := true$
else
    $token\_present := false$
end if
$next, child := nil$; $counter\_number := nil$
if node $id = 0$ then
    $parent := nil$; $token\_counter[i] := i, i = 0, 1, ..k - 1$
    $counter\_number := 0$; $is\_coordinator := true$
end if

Procedure Request_CS
$requesting\_cs := true$
if $token\_present = false$ then
    Send Message_request($i$) to $parent$ /* $i$ refers to self identifier */
    $parent := nil$
end if

Procedure Release_CS
$requesting\_cs := false$
if $child \neq nil$ then
    Send Message_token to $child$
    $token\_present := false$; $child := nil$
end if

Procedure Process_Request($r$)
if $parent \neq nil$ then
    Send Message_request($r$) to $parent$
else
    if $is\_coordinator = false$ then
        $next := r$
    else
        Send Message_child($r$) to $token\_locations[counter\_number]$
        $token\_locations[counter\_number] := r$
        $counter\_number := (counter\_number + 1)\%k$
        $a := token\_locations$; $b := counter\_number$
        $is\_coordinator := false$
        Send Message_token_locations($a$,$b$) to $r$
        $counter\_number := nil$
    end if
end if
$parent := r$

Procedure Process_Token
$token\_present := true$

Procedure Process_Child($r$)
if $requesting\_cs = true$ then
    $child, parent := r$
```

Variable	Type	Description
$token\_present$	boolean	$token\_present$ is true if the node owns the token. Otherwise, it is false.
$requesting\_cs$	boolean	$Requesting\_cs$ is true if the node has invoked the critical section and remains true until it has released the critical section.
$parent$	integer with range $0, 1....N$	$parent$ indicates the node to which requests for critical section execution should be forwarded.
$next$	integer with range $0, 1....N$	$next$ indicates the node which is to be assigned to one of the k token queues after the current node has been assigned to a token queue.
$child$	integer with range $0, 1....N$	$child$ indicates the node to which access permission should be forwarded after the current node leaves the critical section.
$token\_locations$	array of integers with range $0, 1....N$	$token\_locations$ contains the tail nodes of the k child queues
$counter\_number$	integer with range $0, 1...K$	The counter_number indicates the queue number where the next requesting node will be directed toward. $counter\_number$ is only stored at the node which has the $token\_locations$ message.
in_child_queue	boolean	in_child_queue is true if the node is in one of the k distributed child queues, and false otherwise
$is\_coordinator$	boolean	$is\_coordinator$ is true if the node has $token\_locations$, and false otherwise.

Table 1. Variables used in the Fair K Mutual Exclusion Algorithm

```
else
    Send Message_token to $r$
    $token\_present := false; parent := r$
end if

Procedure Process_Token_Locations($a$,$b$)
$token\_locations := a; counter\_number := b$
$is\_coordinator, in\_child\_queue := true$
if $next \neq nil$ then
    $x := token\_locations[counter\_number]$
    Send Message_child($next$) to $x$
    $token\_locations[counter\_number] := next$
    $counter\_number := (counter\_number + 1)\%k$
    $a = token\_location; b = counter\_number$
    $is\_coordinator = false$
    Send Message_token_locations($a$,$b$) to $next$
    $counter\_number := nil; parent := next; next := nil$
end if
```

We now present an explanation of the individual steps of our algorithm.

Request_CS: This procedure is invoked by a node when it wants to enter the critical section. If the node has the token, it can immediately enter the critical section. Otherwise it sends its request Message_request(i) to its parent in the dynamic tree structure.

Release_CS: Node i sets its $requesting\_cs$ to false. It forwards the token to its $child$ (see Table 1) in the distributed queue. If such a node does not exist, the token stays with node i.

Process_Request(r): This function is called when a node receives a request for critical section. If the node i is not root, it will send this request to its $parent$ (see Table 1). If this node is a root, two scenarios are possible. Firstly, if the root node is the coordinator, then it can assign this request to one of the k queues depending on $counter\_number$ (coordinator has this information). Secondly, if the root node is not currently the coordinator (it is in transit), $next$ is set to node r. On reception of the Message_Token_locations, node i can assign its next to one of the k distributed child queues.

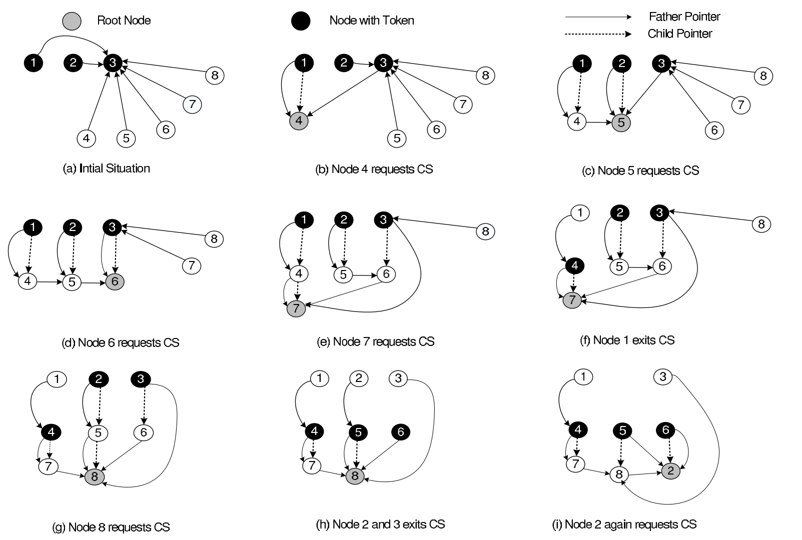
Process_Token: This function is called when a node receives a token. It sets its $token\_present$ as true, and then enters the critical section.

Process_Child(r): On the reception of the message Message_child, if node i has not released the critical section, then it sets its $child := r$. When node i releases the critical section, it forwards the token to node r. In case this message is received when $requesting\_cs = false$, node i will immediately forward the token to node r. In both cases, node i sets its $parent$ to node r. Updating $parent$ pointers on the reception of child messages leads to a reduction in the distance of this node from the root.

Process_Token_locations(a, b): On reception of the message Message_token_locations, a node becomes the coordinator. Node i's $next$ is assigned to a distributed queue by sending Message_child to the tail of the distributed queue (indexed by $counter\_number$), and updated Message_token_locations is sent to it.

3.3. Example

Now we provide an example to explain the underlying principle of the algorithm. In the algorithm, a distributed privileged queue (next queue) is built if the rate of request is high. Moreover, whenever a node joins one of the k distributed queues (child queues), next pointers are reset. However, we only show the child queue structure assuming the rate of request to be low. The figure does not change much in the presence of the next queue. There will be an



(a)

Figure 1. Example

additional distributed queue with head node as the coordinator and tail node as root. Moreover, in finite time, next queue will be reduced and the nodes will be added to the child queues; i.e., root will eventually become the coordinator. In this example the *root node and the coordinator node are the same*.

There are 8 nodes in the system with nodes 1, 2 and 3 having tokens (k=3). All nodes have 3 as their $parent$ except the node 3 i.e., node 3 is the root of the system. Node 3 also acts as a coordinator. In other words it contains the information about the tail nodes of the k child queues ($token\_locations$). In the initial system (Figure 1(a)) child queues are of length 1 and node 3 stores (1, 2 ,3) as the tail nodes of the k child queues.

Now, suppose node 4 wants to access the critical section. It sends a request to its $parent$, which is node 3. Since node 3 is the coordinator, when it receives the request it sends the Message_child to node 1. Node 1 is selected because the $counter\_number$ is initialized to 0. Node 3 updates its $parent$ to 4 and sends the updated $token\_location$ and incremented $counter\_number$ to the new coordinator (node 4). Node 1 receives the Message_child and updates its $child$ and $parent$ to node 4. Node 4 is the new root of the tree. On the reception of the Message_token_location, it becomes the coordinator as well. Figure 1(b) depicts the change.

Figures 1(c),1(d) and Figure 1(e) show the state of the system after node 5, node 6 and node 7 requested for critical section respectively. We can see that the child queues build up. Now, node 1 releases its critical section. It sends the token to $child$ node 4, and resets $child$ to nil. Node 4 receives the token and enters the critical section as shown in Figure 1(f).

Next, node 8 requests for critical section. Also, nodes 2 and 3 release their critical section. Node 2 sends the token to its $child$ node 5 and node 3 sends the token to its $child$ node 6. Both nodes update their $child$ to nil. Nodes 5 and 6 receive the token and enter the critical section. This scenario is shown in Figure 1(g) and Figure 1(h). Finally, node 2 requests for critical section. Figure 1(i) depicts the change.

3.4. Proof of Correctness

We now present a proof of safety as well as liveness for our fair k mutual exclusion algorithm. We assume that there are no failures.

Safety: Nodes $1..k$ are the only k nodes that have the tokens when the system is initialized. Moreover, we are not creating new tokens in our algorithm. Since there are no failures, safety is guaranteed.

Liveness: Consider a node i requesting entry to the critical section. The request of i is transmitted, by the arcs corresponding to $parent$, to a node j for which $parent = nil$. If j is not the coordinator and is waiting for the token, i will be the $next$ of node j. Once the node j becomes the coordinator it adds node i to one of the k child queues and makes node i as the new coordinator by sending the Message_token_locations.

We shall make use of the following five properties in our proof. Due to lack of space, we skip the proofs of these properties

Property 1: If a root requests for critical section, it must have the token.

Property 2: A request for critical section reaches a root in finite time.

Property 3: A node in the token queue obtains the token in a finite time.
Property 4: If a node is in the privileged queue, it will become the coordinator in finite time.
Property 5: If a root is not in the privileged queue, it will enter the privileged queue in finite time.

THEOREM 1 (Proof of Liveness): If a node invokes the critical section, it will be able to enter it within a finite delay.
Proof: Assume that a node i invokes entry to the critical section. If node i is a root, then by Property 1, it already has the token and can enter the critical section. If node i is not the root, its request for token propagates via parent pointers to a root in finite time (by Property 2). The root could be in the one of the two possible states:
(a) Root is the coordinator: If the root is the coordinator, as a result of receiving Message_request(i), it sends Message_child to one of the tails of child queues. Node i therefore becomes part of a child token queue, and by Property 3, it will get the token in finite time.
(b) Root is not the coordinator: Two scenarios may arise:

(i) Root is not in the privileged queue: The root will be in the privileged queue in finite time (by Property 5).

(j)Root is in the privileged queue: Then the root will become coordinator in finite time using Property 4. It will then send Message_child to one of the tails in the child queues, and by Property 3, node i will get the token in finite time.

4. Experimental Results

In this section, we present simulation results for our algorithm. Our metrics of interest are the following:
1) Mean time to get the token (mttt): This is average access time, and should be small.
2) Maximum time to get the token: This metric is an indicator of the fairness of an algorithm. The more fair the algorithm, the closer this value should be to mttt.
3) Average number of messages per critical section entry: A smaller average number of messages is desirable for scalable p2p solutions.

We have implemented our algorithm inside an event-based simulator [1]. The mutual exclusion requests arrive according to a Poisson process with parameter λ, the rate of arrival. The critical section duration ($cset$) is set to 10 seconds. This relatively higher value is reasonable as in our applications, a node needs to establish a TCP connection with a server and transfer its data (log file). We compare our algorithm to the k mutual exclusion algorithm in [2], labeled as BV95. We limit our comparative analysis to BV95 as other protocols for k mutual exclusion have worse mean access times and suffer from very high message overhead.

Our simulation considers two settings. In the first LAN setting, the latency (T) between every pair of nodes is 1 second. In the second WAN setting, we estimate the latencies between each pair of nodes using the King data set available at [5]. This data set contains measured latencies between Internet domain name servers (DNS) and is highly heterogeneous. The average round trip time (RTT) in the data set is around 182 ms and the maximum RTTs are around 800 ms. We simulate our protocol for $N = 100$ nodes and $k = 3$ tokens.

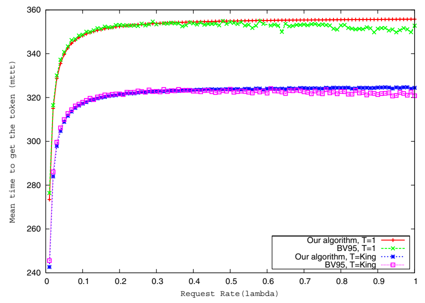


Figure 2. Plot of Mean Time to token vs request rate with constant latency 1 and with latencies estimated from King Data Set

In Figure 2, we show the variation in the mean time to get the token (mttt) for different values of λ for both settings. Our protocol performs well, even for high values of request rate. The mttt values stabilize at 355 seconds beyond $\lambda = 0.5$. Further, for all values of λ, our protocol performance is comparable to BV95. We note that the mttt trends in the two settings are the same. Using the latency values in the King data set does not have a huge impact on the results due to the dominating effect of a large critical section executing time. Thus, we have been able to achieve the same level of performance in terms of mttt as [2], despite using only a single instance of the forest tree structure.

The average contention in the system can be computed as the product of the average arrival rate and the average time a node spends in the system, as given by Little's law [7]. When $\lambda = 1$, we have that the average contention is $C = N \cdot \frac{1}{mttt+cset+1} \cdot (mttt+cset)$, which is approximately equal to N. Moreover, optimal mttt is given by $\frac{\lambda \cdot N \cdot cset}{k}$. In the LAN setting, our algorithm has a mean mttt of 355s, which is close to the optimal value of 333s.

To measure the fairness, we plot the maximum time for a node to get the token. Figure 3 shows, for the LAN setting, the behavior of the globally maximum time to get the token for varying values of λ. We performed 100 trials of our experiment and in each experiment, a node requests for CS 2000 times. We can see that for $\lambda = 0.5$, the maximum time to get the token for our algorithm is as much as 370 seconds, while it is approximately 1580 seconds for BV95. Thus, our protocol outperforms BV95 by more than an order of magnitude in terms of fairness characteristics. It is also interesting to note that unlike BV95, the worst case time of our protocol does not deviate much from the mean time to get the token.

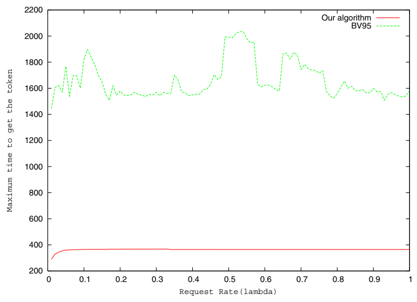


Figure 3. Globally maximum time to get the token vs request rate

It may well be the scenario that the poor performance of BV95 is due to a few nodes which took a long time to get the token. Thus,we study the cumulative distributions of the maximum time to get the tokens for both the algorithms. We fix $\lambda = 0.5$ for this experiment, and perform simulations for the LAN setting. Figure 4 shows the CDF of the maximum time to the get token for our algorithm and BV95. We can see that the maximum time to get the token is very close to the mean value of about 370 seconds for all the nodes, with the deviation from the mean (access time spread) being less than 15 seconds. On the other hand, we can see that all the nodes have their worst access times greater than 1300 seconds, and the $90th$ percentile is over 1700 seconds for the maximum access time of BV95, with the access time spread being more than 1250s. This analysis demonstrates the fairness of our algorithm.

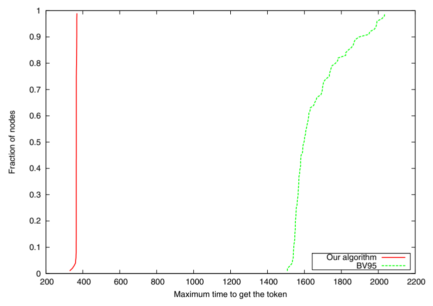


Figure 4. Plot of cumulative distribution of the maximum time to get the token for our algorithm and for [2]

Next, Figure 5 shows the plot of average number of messages sent per critical section entry for varying values of λ. This is relatively high for our protocol because of two reasons: 1) New messages in our algorithm such as Message_child and Message_token_locations, compared to BV95, and 2) To preserve the fairness properties, we do not cache requests for the critical section.

Note that BV95 uses k instances of a dynamic tree, and simple modifications like not caching requests for critical section access or introducing new messages would still result in poor fairness as compared to our algorithm. This is because in order to send a request for critical section access, there is uncertainty in choice of the optimal dynamic tree to use. Our algorithm intuitively solves this problem by using only a single instance of a dynamic tree. Moreover, BV95 does not permit the tradeoff between increase in message complexity and fairness, thus its latency cannot be reduced by increasing its bandwidth.

We conclude that at the cost of sending higher number of messages per critical section entry, our algorithm achieves similar performance for mttt values and an order of magnitude performance improvement for maximum time to get the token as compared to BV95.

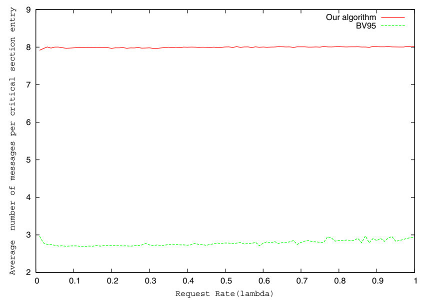


Figure 5. Plot of Average Messages vs request rate

5. Fault Tolerant Methodology

We use a fault tolerant methodology for structured p2p networks like Chord [20]. Chord has the concept of successors and predecessors at each node. The main idea of the fault tolerance methodology is that the group of successors of a node will act as replica (back-up) nodes for the same. The successor nodes of a node i store the state information of node i. To ensure data consistency amongst all backup nodes, whenever the node i changes its state, it initiates an atomic broadcast to all the successors in the replica set. Now, if the node i fails, all messages directed to the identifier of node i will be received by the successor of node i. This is because in p2p systems like Chord, a node owns the id space between itself and its predecessor, and all messages for that id space are directed to that node. Thus when a node i fails, one of its successors (say node j) in the replica set will receive the message. Since node j has the state information about node i, it can process the message. Node j also initiates an atomic broadcast to the replica set of node i to update node i's state.

Given the churn in p2p systems (continuous node arrival and departure), there may be a scenario where a node and all its replicas fail. Such a scenario can be detected when a node receives a message destined to an identifier for which it is the owner, but it does not have the state information of that identifier. Since the node cannot process the message, it resorts to conventional fault tolerant algorithms such as those described in [18]. The conventional algorithms involve the use of broadcasts, and are thus very expensive. Note that the probability of p back up nodes failing simultaneously is f^p, where f is the probability of a single node failure. The probability of all backup nodes failing simultaneously is quite low for $p = 6$ and $f = 0.01$, the probability is 10^{-12}.

We also study the success of the fault tolerant methodology under catastrophic failures (to minimize the broadcasts used in conventional fault tolerance). We have assumed that the Chord entries satisfy the specifications given in [20]. Simulations are done for $\lambda = 0.5$ in the LAN setting. Figure 6 shows the success probability of our protocol maintaining safety and liveness when 20% and 50% random nodes fail simultaneously. We can see that for 20% node failures, the use of $p = 4$ results in a success probability greater than 0.95. For 50% node failures, the use of $p = 6$ backup nodes results in a success probability greater than 0.8.

We conclude that that using Chord successors to maintain state information can obviate the need for expensive broadcast based fault tolerance.

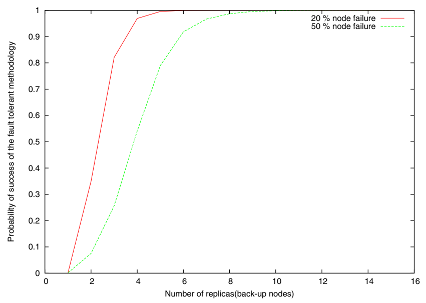


Figure 6. Probability of success vs number of back-up nodes under catastrophic failures

6. Conclusion

In this paper, we have proposed and evaluated a token based fair k mutual exclusion algorithm for p2p systems. We used a novel technique inspired by a real world scenario to provide fair k mutual exclusion using a single forest tree structure. Our comparative analysis with [2] shows that our algorithm achieves an order of magnitude improvement in fairness characteristics, while providing low spread of access time. This is done at the cost of a slight increase in the message overhead. Our analysis also shows that the algorithm is also resilient to failures.

References

[1] J. Banks, J. Carson, et al. *Discrete-event system simulation*. Prentice Hall, 2001.

[2] S. Bulgannawar and N. Vaidya. A distributed k-mutual exclusion algorithm. *Proc . ICDCS*, pages 153–160, 1995.

[3] Gridftp: http://www.globus.org/grid_software/data/gridftp.php.

[4] J. Helary, A. Mostefaoui, and R. IRISA. A $o(\log_2 n)$ fault-tolerant distributed mutual exclusion algorithm based on open-cube structure. *Proc . ICDCS*, pages 89–96, 1994.

[5] King data set: http://pdos.csail.mit.edu/p2psim/kingdata.

[6] S. Lin, Q. Lian, M. Chen, and Z. Zhang. A practical distributed mutual exclusion protocol in dynamic peer-to-peer systems. *Proc . IPTPS*, 2004.

[7] J. Little. A Proof for the Queuing Formula: L= λ W. *Operations Research*, 9(3):383–387, 1961.

[8] S. Lodha and A. Kshemkalyani. A fair distributed mutual exclusion algorithm. *IEEE TPDS*, 11(6):537–549, 2000.

[9] M. Maekawa. A sqrt n algorithm for mutual exclusion in decentralized systems. *ACM TOCS*, 3(2):145–159, 1985.

[10] M. Muhammad. Efficient mutual exclusion in peer-to-peer systems. Master's thesis, UIUC, 2005.

[11] M. Naimi, M. Trehel, and A. Arnold. A log(n) distributed mutual exclusion algorithm based on path reversal. *JPDC*, 34(1):1–13, 1996.

[12] K. Park and V. Pai. Scale and performance in the coblitz large-file distribution service. *Proc . NSDI*, pages 3–3, 2006.

[13] The planetlab global research network: http://www.planet-lab.org/.

[14] K. Raymond. Multiple entries with ricart and agrawalas distributed mutual exclusion algorithm. Technical Report 78, University of Queensland, 1987.

[15] K. Raymond. A tree-based algorithm for distributed mutual exclusion. *ACM TOCS*, 7(1):61–77, 1989.

[16] G. Ricart and A. Agrawala. An optimal algorithm for mutual exclusion in computer networks. *Communications of the ACM*, 24(1):9–17, 1981.

[17] K. Sheers. HP OpenView Event Correlation Services. *Hewlett-Packard Journal*, 47(5):31–42, 1996.

[18] J. Sopena, L. Arantes, M. Bertier, and P. Sens. A fault-tolerant token-based mutual exclusion algorithm using a dynamic tree. *Proc . EuroPar*, pages 654–663, 2005.

[19] P. Srimani and R. Reddy. Another distributed algorithm for multiple entries to a critical section. *IPL*, 41(1):51–57, 1992.

[20] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *Proc . SIGCOMM*, 31(4):149–160, 2001.

[21] I. Suzuki and T. Kasami. A distributed mutual exclusion algorithm. *ACM TOCS*, 3(4):344–349, 1985.

[22] R. van Renesse, K. Birman, D. Dumitriu, and W. Vogels. Scalable Management and Data Mining Using Astrolabe. *Proc . IPTPS*, 2002.

[23] P. Yalagandula and M. Dahlin. A scalable distributed information management system. In *Proc . SIGCOMM*, Aug. 2004.