HAL open science

Section critique à entrées multiples tolérante aux fautes et utilisant des détecteurs de défaillances

Mathieu Bouillaguet, Luciana Arantes, Pierre Sens

▶ **To cite this version:**

Mathieu Bouillaguet, Luciana Arantes, Pierre Sens. Section critique à entrées multiples tolérante aux fautes et utilisant des détecteurs de défaillances. [Research Report] 2007, pp.10. inria-00128988v1

HAL Id: inria-00128988

https://inria.hal.science/inria-00128988v1

Submitted on 5 Feb 2007 (v1), last revised 9 Jun 2008 (v3)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Fault Tolerant K-Mutual Exclusion Algorithm Using Failure Detector

Mathieu Bouillaguet, Luciana Arantes, and Pierre Sens

LIP6, Université Pierre et Marie Curie, INRIA
{mathieu.bouillaguet,luciana.arantes,pierre.sens}@lip6.fr

Abstract. This paper presents a fault tolerant k-mutual exclusion algorithm built on top of an unreliable failure detector. The algorithm is an extension of the Raymond's algorithm [Ray89] where the number of nodes n is dynamically adapted according to the information provided by the underlying failure detector. Compared to the initial algorithm, our proposal tolerates $n-1$ failures without restricting the number of concurrent accesses to the shared resource. Simulation studies show the good performance of our algorithm compared to the originals' one when failures are injected.

1 Introduction

Mutual exclusion is a well-known problem to coordinate accesses of multiple processes to a shared resource. The k-mutual exclusion problem is a generalization of the former which allows at most k processes to get one unit of the shared resource simultaneously. It involves then a set of n processes each of them requesting access to one of the k units of the resource, called the critical section (CS). Therefore, at most k processes can be in the CS at a given time. A k-mutual exclusion algorithm satisfies the *safety property* and the *liveness property* by respectively assuring that at most k processes are in the *CS* at the same time and that every critical section request is eventually satisfied.

Distributed k-mutual exclusion algorithms can basically be divided into two groups: permission-based [Ray89], [PMP+96], [HJK93] and token-based [SR92], [MBB+92], [BV95]. The first group of algorithms is based on the principle that a node gets into critical section only after having received permission from all or a subset of the other nodes of the system. In the second group of algorithms, the possession of the single token or one of the tokens gives a node the right to enter into the critical section. The latter usually presents an average lower message cost of messages, but is less fault tolerant than permission-based algorithms which, by using broadcast, are naturally more relisient to failures.

Raymond k-mutual exclusion algorithm [Ray89] is an extension of Ricart-Agrawala's [RA81] permission-based 1-mutual exclusion algorithm. When a node wants to enter the CS, it broadcasts a message to the other $(n-1)$ nodes. The requesting node can enter the critical section if no more than $k-1$ of the other $n-1$ nodes are currently executing the CS i.e., only after having gathered $n-k$ permissions sent from other nodes.

Even if Raymond's algorithm does not consider node failure, the fact that it does not need to wait for a permission from all other participants implicitly renders it fault tolerant to some extent. It tolerates up to $k-1$ faults. In other words, if instead of executing the *CS*, $k-1$ nodes were crashed, a node asking to execute a *CS* would still achieve to get it. However, each crash reduces the effectiveness of the algorithm since the number of processes that can concurrently execute the critical section decreases by one. Thus, by providing information about nodes' aliveness, we propose in this paper an extension of Raymond's algorithm which makes it more efficient than the original one if nodes crash. Furthermore, it tolerates $n-1$ nodes crash instead of $k-1$. Notice that in Raymond's algorithm, no process would be able to enter the critical section beyond $k-1$ failures, contrary to our's that goes on assuring that at most k processes can concurrently execute the CS till $n-1$ crashes. In order to give information to a process about the aliveness of the other processes we use an unreliable failure detector in our solution.

An unreliable failure detector (FD) [CT96] is a well-known basic block which offers information about process failures. It can be informally considered as a per process oracle, which periodically provides a list of processes that it currently suspects of having crashed. It is unreliable since it can make mistakes. Our approach is based on unreliable detectors of class $\mathcal{T}$ [DGFGK05] since it is the weakest one to solve the fault-tolerant 1-mutual exclusion problem. Thus, a per process failure detector $\mathcal{T}$ module will periodically provides information to the corresponding process about the actual state of the system. Such information allows each process to dynamically updates its knowledge of the actual number of running nodes providing a more efficient execution of the algorithm when compared to the original's one.

In the rest of this paper, we consider an asynchronous message passing system model. The number of nodes is n and the set of participants is known by all of them. Nodes can only fail by crashing, and crashes are permanent. Communication channels are reliable, but messages might be delivered out of order. The number of copies of the resource is k. As we consider that there is one process per node, the words node and process are interchangeable.

The paper is organized as follows. Section 2 presents several classes of failure detectors. Section 3 briefly describes the Raymond's algorithm. Section 4 presents our fault-tolerant algorithm while section 5 outlines the proof. Simulation performance results are shown in Section 6. Finally, Section 7 concludes the paper.

2 Failure detectors classes

Chandra and Toueg [CT96] have introduced the concept of unreliable failure detectors which provide information about processes that might have crashed. Each process is equipped with a local module of failure detector which outputs a list of processes it currently suspects to be faulty. Later, if it believes that a

previously suspected process is still alive it can remove it from its list. Therefore, each module may repeatedly add and remove processes from its list of suspects.

Failure detector (FD) are abstractly characterized by the *completeness* and *accuracy* properties which provide different classes of FD: *completeness* and *accuracy. Completeness* characterizes the failure detector capability of suspecting every incorrect process permanently. *Accuracy* characterizes the failure detector capability of not suspecting correct processes. The strongest failure detector is the perfect failure detector $\mathcal{P}$. It is characterized by *strong completeness* and *strong accuracy*, which means that every crashed process is eventually suspected permanently by every correct process and no process is suspected before it crashes. A weaker failure detector is $\mathcal{S}$ which relaxes the accuracy property to *weak accuracy* which means that *some* correct process is never suspected.

Delporte-Gallet and al. [DGFGK05] have introduced the failure detector of class $\mathcal{T}$, also called the *trusting* failure detector. They proved that it was the weakest failure detector to solve the fault-tolerant 1-mutual exclusion problem. This failure detector has the *strong completeness* property and satisfies the following accuracy properties:

Eventual strong accuracy: There is a time after which correct processes are not suspected by any correct process.

Trusting accuracy: Every process j that is suspected by a process i after being trusted (i.e. not suspected) once by i is crashed.

The failure detector $\mathcal{T}$ is strictly weaker than the failure detector $\mathcal{P}$. Roughly speaking, failure detector $\mathcal{T}$ can temporarily suspect a correct process, as long as it had never been removed from the list of suspects.

3 Raymond's algorithm

In Raymond's algorithm [Ray89], when a node i wants to enter its critical section, it broadcasts a $REQUEST$ message to the other $(n-1)$ processes. Each request is timestamped with Lamport's logical clock (sequence number + identity of the node)[Lam78]. Upon receiving such message, a node j which is not requesting a resource, immediately gives its permission to i by sending it back a $REPLY$ message. If j is in a critical section, it defers sending its permission until it exits the CS. In the case where j is also requesting a resource, the sequence numbers of both requests are compared. If they are equal, the identity of the nodes breaks tie. If i's request takes priority, j sends it back a REPLY message, otherwise j defers it till it releases the CS. When i has gathered $(n-k)$ permissions it enters its CS since it is certain that no more than $(k-1)$ of the other nodes are currently executing the critical section, which ensures the *safety* property.

The timestamp of request messages guarantees the *liveness* property of the algorithm since it defines a total order for the pending requests.

As previously said, Raymond's algorithm implicitly tolerates $k-1$ crashes i.e., the *safety* property still holds until up to $k-1$ failures occur. Thus, even if one or more nodes crash a second node is still able to access a copy of the

resource if it can collect $(n - k)$ permissions. On the other hand, each time a failure occurs, the maximum number of processes that can concurrently execute the critical section decreases by one, reducing the effectiveness of the algorithm.

4 Solving k-mutual exclusion using failure detector

Raymond's algorithm has no information about node crashes i.e., the number of participants is set to n at the initialization phase and does not change. In order to provide such information, we use in our solution failure detectors of class $\mathcal{T}$.

The trusting accuracy property of failure detectors $\mathcal{T}$ guarantees that suspected nodes that were previously trusted are in fact crashed. Thus, the failure detector $\mathcal{T}$ will inform the nodes of the actual state of the system. When a node learns that another node has crashed, either from its local failure detector module or by receiving a $CRASH$ message, it decrements its local variable n. Hence, contrary to Raymond's algorithm, the number of $REPLY$ messages $(n - k)$ needed by a requesting process will be reduced by one.

4.1 Description of the algorithm

Algorithm 1 is the complete pseudo-code of our fault tolerant k-mutual exclusion algorithm. It is made up of an *Initialization* procedure (l. 1-8), a $Request_resource$ procedure (l. 9-16), a $Release_resource$ procedure (l. 17-20), a set of event handlers (l. 21-42) which treat the reception of the different types of message, and the per process failure detector $\mathcal{T}$ module (l. 43-45).

There are five types of messages in our algorithm : $REQUEST$ messages are broadcast by a process which executes the $Request_resource$ procedure in order to inform the other processes that it wants to access a resource. Requests carry the identity of the sender and the current value of the local logical clock; $REPLY$ messages are permissions tickets sent by processes in response to a $REQUEST$ message. Multiple permissions can be aggregated in a single $REPLY$ message by carrying an additional counter indicating the number of deferred replies that the message includes. The $INIT$ message is a trust request sent once by each process during the initialization phase. When a process receives such a message, it acknowledges its reception by returning an ACK message. Finally, $CRASH$ messages are broadcast when a process detects a crash of another process to inform the other participants of the process failure.

Each process i keeps a set of local variables: H_i, the value of the logical clock; $last_i$, the value of the logical clock when the last message $REQUEST$ was sent by i; and $perm_count_i$, the number of permissions received. To prevent a $REPLY$ message to an earlier request to be considered as a reply to the current request, a $reply_count_i$ array keeps track of the number of outstanding reply messages still to come from all the other nodes. The $defer_count_i$ array is used to keep track of the number of deferred requests to be replied later for each node. Additionally, a $trusted_i$ and a $crashed_i$ set is used by node i to respectively save the set of nodes that it once trusted and the set of crashed ones. Finally nodes

can interrogate their local failure detectors $\mathcal{T}$ and $\mathcal{S}$, which provide a list of currently suspected processes through respectively the $\mathcal{T}_i$ and $\mathcal{S}_i$ sets.

When a node i requests a resource, it broadcasts a $REQUEST$ message, it increments $reply_count[j]$ for each node $j \neq i$ and it waits for $n - k$ replies before entering the CS (l. 12-15). Upon reception of a $REQUEST$ message, node j updates its logical clock and it sends back a $REPLY$ message (l. 27) only if it is not in the CS or if its current request hasn't priority over i's one. Otherwise, it defers the request (l. 25). When i receives a $REPLY$ message from j it decrements $reply_count_i[j]$. If j has replied to all the previous requests sent by i (l. 31), then $perm_count$ is incremented. Upon exiting the CS, node i replies to the deferred requests (l. 17-20).

If a node crashes, at least one process will execute line 43-45 broadcasting a $CRASH$ message. When a $CRASH$ message is received, the number of alive nodes is decremented (l. 42). However if the node had previously replied to the node's request its permission is canceled (l. 41).

The *Initialization* procedure is executed once by each process at the beginning of the algorithm. The condition of line 8 and the use of failure detector $\mathcal{S}$ is justified by the necessity that at least one correct process trusts each process at the end of the initialization. It ensures that at the end of the initialization, each node is included in at least one *trusted* set, and that condition line 43 will be verified at one node.

```
 1: state_i := not_requesting                              ▷ Initialization
 2: H_i := 0; last_i := 0
 3: perm_count_i := 0
 4: reply_count_i[N] := 0
 5: defer_count_i[N] := 0
 6: trusted_i := ∅; crashed_i := ∅
 7: send INIT(i) to all
 8: wait until receive ACK from all j ∉ S_i

    Request_resource():                                    ▷ Node wishes to enter CS
 9: state_i := requesting
10: last_i := H_i + 1
11: perm_count := 0
12: for all j ≠ i do
13:     send REQUEST(i, last_i) to j
14:     reply_count_i[j] + +
15: wait until (perm_count_i ≥ n − k)
16: state_i := CS

    Release_resource():                                    ▷ Node exits the CS
17: state_i := not_requesting
18: for all (j ≠ i : defer_count_i[j] ≠ 0) do
19:     send REPLY(i, defer_count_i[j]) to j
20:     defer_count_i[j] := 0
```

```
21: upon receive REQUEST(j,k) do                    ▷ A REQUEST message is received
22:     H_i := max(H_i, k)
23:     if (j ∉ crashed_i) then
24:         if (state_i = CS) or (state_i = requesting and (last_i, i) < (k, j)) then
25:             defer_count_i[j] + +
26:         else
27:             send REPLY(i, 1) to j

28: upon receive REPLY(j,x) do                        ▷ A REPLY message is received
29:     if (j ∉ crashed_i) then
30:         reply_count_i[j] := reply_count_i[j] − x
31:         if (state_i = requesting) and (reply_count_i[j] = 0) then
32:             perm_count_i + +

33: upon receive INIT(j) do               ▷ A trust request is received from j ∈ Π
34:     wait until j ∉ T_i
35:     trusted_i := trusted_i ∪ {j}
36:     send ACK(i) to j

37: upon receive CRASH(j) do                          ▷ A CRASH message is received
38:     if (j ∉ crashed_i) then
39:         crashed_i := crashed_i ∪ {j}
40:         if (state_i = requesting) and (reply_count_i[j] = 0) then
41:             perm_count_i − −
42:         n − −

43: upon (j ∈ trusted_i and j ∈ T_i) do              ▷ A crash of process j is detected
44:     trusted_i := trusted_i − {j}
45:     send CRASH(j) to all
```

Algorithm 1: Raymond's extended algorithm

4.2 Example of execution

Figure 1 depicts a possible execution of our extended algorithm. The system consists of 4 processes and 2 resources. At T_0, node 2 has exclusive access to the first resource. Node 1 requests then a resource at T_1 by broadcasting a $REQUEST$ message to all the other processes. At T_2, 3 sends a $REPLY$ message to 1 but node 4 crashes. At T_3 the condition ($4 \in trusted_3$ and $4 \in \mathcal{T}_3$) (l. 43) is verified and node 3 broadcasts a $CRASH$ message. At time T_4, node 1 receive the $CRASH$ message sent by 3 and executes lines 37 to 42 of the algorithm. The number of alive nodes becomes 3 and the condition to enter critical section ($perm_count_3 \geq 3-2$) is now verified (l. 15), node 1 executes its critical section.

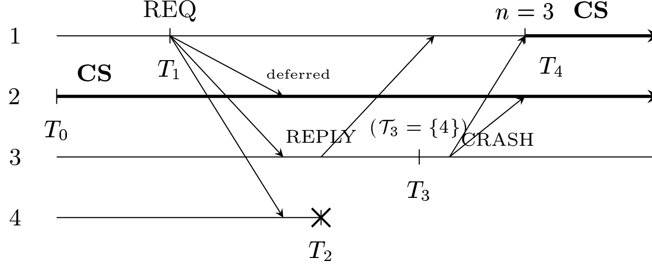

Fig. 1: Example of an execution of our extended algorithm with $n = 4$ and $k = 2$

5 Outline of proof

We must prove that our extension of Raymond's algorithm satisfies the *safety* and *liveness* properties. Notice that in our approach, we consider that processes do not crash before the initialization phase. In case of process crashes before the end of the initialization, the *safety* property would still be ensured until up to k-1 failures.

5.1 Safety

Lemma 1. *No more than k different processes are in their critical section at the same time*

Proof. Suppose not. Assume that at time t_c, $m > k$ nodes are executing their critical section. Consider the pairs (S,N)= (sequence number, node identity) included in the $REQUEST$ messages and used by the m nodes that gained access to the critical section. These pairs form a total order. Hence, the nodes in critical section can be labelled $N_1, \ldots, N_k, N_{k+1}, \ldots, N_m$ so that $(S_{N_1}, N_1) < \cdots < (S_{N_k}, N_k) < (S_{N_{k+1}}, N_{k+1}) < \cdots < (S_{N_m}, N_m)$. Consider the node N_{k+1}. To enter critical section N_{k+1}, received $(n-k)$ $REPLY$ messages from the other $(n-1)$ nodes. Or rather, at most $k-1$ nodes did not send a reply to N_{k+1}. Thus, of the k nodes $N_1, \ldots, N_k$ one of them $N_{X(\leq k)}$ sent a reply to N_{k+1}. Consider the situation when N_X received the $REQUEST(S_{N_{k+1}}, N_{k+1})$. There are 4 cases:

- *Case 1.* N_X is in the state *not_requesting* or *requesting* with sequence number $(S_{N_X}, N_X) > (S_{N_{k+1}}, N_{k+1})$. Upon receiving the $REQUEST$ message, H_X became $\geq S_{N_{k+1}}$, hence N_X could not be executing the critical section at time t_c with $(S_{N_X}, N_X) < (S_{N_{k+1}}, N_{k+1})$
- *Case 2.* N_X is in the state CS or *requesting* with sequence number $(S_{N_X}, N_X) < (S_{N_{k+1}}, N_{k+1})$. In this case, N_X would defer replying to N_{k+1}.
- *Case 3.* N_X is executing or attempting to execute the critical section on a previous occasion with sequence number R such that $(R, N_X) \leq (S_{N_X}, N_X) < (S_{N_{k+1}}, N_{k+1})$. Hence S_{N_X} would become $\geq S_{N k+1}$ and so N_X could not be executing the critical section at time t_c with $(S_{N_k}, N_k) < (S_{N_{k+1}}, N_{k+1})$.

– *Case 4.* N_x crashes. Obviously it can't reply to N_{k+1}.

Thus it is impossible for any node $N_{X(\leq k)}$ to reply to the request of node N_{k+1}. □

5.2 Liveness

Lemma 2. *If a correct process requests a unit of the resource, and it has the most priority request then at some time later it obtains the resource.*

Proof. Suppose that a correct process i is requesting a unit of the resource at some time t_c with $last_i = l_i$, its request has priority over all the others, and process i is never in its critical section after t_c. By the algorithm, i never reaches line 16. Thus i is blocked at a "wait" clause either at line 8 or line 15. The first "wait" clause (line 8 of algorithm 1) is not able to block the process due to the strong completeness property of $\mathcal{S}$. Eventually all processes not in $\mathcal{S}_i$ are correct, so these processes not in $\mathcal{S}_i$ eventually receive the $INIT$ message of i. By the weak accuracy property of $\mathcal{S}$, there is at least one correct process that is never suspected. So i wait for the reply of at least one correct process. These processes, receive a $INIT$ message from i and execute lines 33 to 36.

By the eventual strong accuracy property of $\mathcal{T}$, every correct process is eventually trusted by all correct processes. Hence, the "wait" clause of line 34 is not blocking, and the processes add i to their *trusted* set and send back a ACK message to i, unblocking the "wait" clause line 8.

Thus i is blocked at the "wait" clause of line 15 having sent a $REQUEST(i, last_i)$ message to all $j \neq i$. Four cases are possible for process j:

(a) Process j is in the state *not_requesting*. The condition line 24 is not satisfied and the process sends a permission (line 27).
(b) Process j is in the state *requesting*. Since i has priority over all the others requests, j sends back its permission.
(c) Process j is in its critical section. The duration of the critical section is bounded so it will eventually send back a reply message to i when executing its **Release_resource()** routine (line 17 to 20).
(d) Process j crashes. By the trusting accuracy property of $\mathcal{T}$, some correct process m eventually and permanently suspect it. In other words, the condition of line 43 is eventually satisfied at some process m for j ($j \in trusted_m$ and $j \in \mathcal{T}_m$). Thus, m sends a $CRASH$ message to all processes and every correct process eventually receive it. Upon receiving the $CRASH(j)$ message, i decrements the number of participating nodes n, and decrements the number of permissions received if it had already received one from j. Thus, the condition line 43 will reflect the new situation, since n represents the number of not crashed processes.

Hence i will eventually receive exactly n replies, with n representing the number of alive processes. But i is blocked at line 15. It's a contradiction. □

Lemma 3. *If a correct process requests a unit of the resource, then at some time later it obtains it.*

Proof. By lemma 2, the process that has priority over the others will eventually obtain a unit of the resource. Once it exits the critical section, the process's request was satisfied and will not be considered anymore. Since requests are totally ordered, each of them will eventually have the highest priority, obtaining then a resource. □

Theorem 1. *The algorithm 1 solves fault tolerant k-mutual exclusion using* $\mathcal{T}$*, in an environment* $\mathcal{E}_f$ *with* $f < n - 1$ *provided that no process crashes before the initialization.*

Proof. The theorem 1 follows directly from lemmas 1 and 3. □

6 Performance comparison

In order to evaluate the efficiency of our algorithm, we developed a simulator. We compared the execution of both algorithms with 15 nodes and 5 resources by measuring the number of resources in use. Both algorithms execute the same scenario. Crash failures are injected during the run (signaled by a triangle on the graph).

In Raymond's algorithm case, each crash clearly decreases the maximum number of concurrent accesses by one. After the k^{nth} crash, no new request can be served; some requests issued before the fifth crash can be served. Our algorithm still progresses until up to $n-1$ process failures. The maximum number of resources that can be concurrently accessed is not bounded by the number of failures, it decreases only due the number of remaining alive processes.

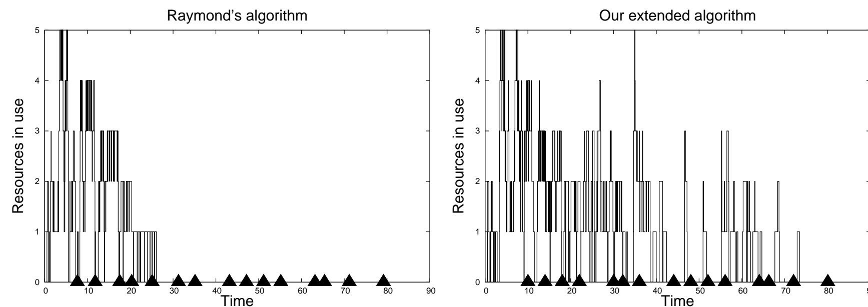

Fig. 2: Efficiency comparison of Raymond algorithm and our extension

The number of messages sent per CS when no crash occurs is equivalent to Raymond's algorithm (bounded by $2n-1$ and $2n-k-1$). When a crash happens, n messages are sent by the node which detects the failure. At the initialization, each process sends once between $n-1$ and $2(n-1)$ messages.

7 Conclusion

We presented a new fault tolerant algorithm that solves the k-mutual exclusion problem. Compared to the classical Raymond's algorithm, our algorithm dynamically adapts the initial number of processes. Thus, the number of concurrent access is increased when failure occurs and we tolerate a maximum numbers of processes failure (n-1). The algorithm also assumes an asynchronous network augmented with the $\mathcal{T}$ failure detector which is the weakest failure detector to resolve the 1-mutual exclusion problem.

References

[BV95] Shailaja Bulgannawar and Nitin H. Vaidya. A distributed k-mutual exclusion algorithm. In *International Conference on Distributed Computing Systems*, pages 153–160, 1995.

[CT96] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the Association for Computing Machinery*, 43(2):225–267, March 1996.

[DGFGK05] Carole Delporte-Gallet, Hugues Fauconnier, Rachid Guerraoui, and Petr Kouznetsov. Mutual exclusion in asynchronous systems with failure detectors. *J. Parallel Distrib. Comput.*, 65(4):492–505, 2005.

[HJK93] S.T. Huang, J.R. Jiang, and Y.C. Kuo. k-coteries for fault-tolerant k entries to a critical section. *Distributed Computing Systems, 1993., Proceedings the 13th International Conference on*, pages 74–81, 1993.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[MBB+92] K. Makki, P. Banta, K. Been, N. Pissinou, and EK Park. A token based distributed k mutual exclusion algorithm. *Parallel and Distributed Processing, 1992. Proceedings of the Fourth IEEE Symposium on*, pages 408–411, 1992.

[PMP+96] Niki Pissinou, Kia Makki, E. K. Park, Z. Hu, and W. Wong. An efficient distributed mutual exclusion algorithm. In *ICPP, Vol. 1*, pages 196–203, 1996.

[RA81] Glenn Ricart and Ashok K. Agrawala. An optimal algorithm for mutual exclusion in computer networks. *Commun. ACM*, 24(1):9–17, 1981.

[Ray89] Kerry Raymond. A distributed algorithm for multiple entries to a critical section. *Inf. Process. Lett.*, 30(4):189–193, 1989.

[SR92] Pradip K. Srimani and R. L. N. Reddy. Another distributed algorithm for multiple entries to a critical section. *Inf. Process. Lett.*, 41(1):51–57, 1992.