Fault Resilience for Distributed Locking

David C. Choweller*
University of California
Riverside, CA 92521-0304
U.S.A.
Telephone: [1] (909) 787-2961
Fax: [1] (909) 787-4643
Email: davec@cs.ucr.edu

Brett D. Fleisch†
University of California
Riverside, CA 92521-0304
U.S.A.
Telephone: [1] (909) 787-7206
Fax: [1] (909) 787-4643
Email: brett@cs.ucr.edu

UCR-CS-96-3
April 18, 1995

Abstract

This paper presents a fault tolerant algorithm for implementing mutual exclusion in distributed systems. The work improves on both the mutual exclusion algorithm used in the Quarks DSM system and Raymond's elegant distributed mutual exclusion algorithm. The algorithm has been implemented and validated in an improved version of Quarks.

*Supported by a grant from Computer Marketplace, Inc.
†Supported by a grant from IBM Corporation, the UC Micro program and Computer Marketplace, Inc.

Contents

1 Overview

Fault tolerance is important in distributed systems because each component of a distributed system can fail independently, and because the large number of components present in such a system makes it more likely that one of them is faulty. In a distributed system, a part of the system may fail, while the rest of the system continues to function. Therefore, software must be prepared to deal with partial failures, and incorrect operation of the system caused by such failures.

The algorithm described in this paper is implemented in a shared memory system. Shared memory is a familiar programming environment to most programmers, allowing different processes access to global data. One form of distributed system, the shared memory multiprocessor, allows processes on different processors to share the same physical memory. A number of such machines have been built, and programs have been written to exploit the parallelism afforded by them. An alternative approach is to use off-the-shelf components to build a network of workstations providing the illusion of a global shared memory. This approach, known as Distributed Shared Memory (DSM)[1][4][6], is more scalable and cost-effective. The mutual exclusion algorithm described in this paper is implemented in an enhanced version of the Quarks Distributed Shared Memory (DSM) system[3].

DSM [1][4] supports a familiar programming abstraction similar to that provided in shared memory parallel processors. The abstraction can extend to a distributed system in a referentially transparent manner. While programming a distributed system can be relatively difficult, programming a DSM system is as easy as programming a shared memory multiprocessor.

1.1 Process Synchronization in Shared Memory Systems

While shared memory systems are convenient to program, synchronization between cooperating processes must be managed explicitly. In particular, shared memory programs require support for synchronization and mutual exclusion among processes. Currently, DSM systems provide a number of mechanisms for this purpose:

- *locks* provide for mutually exclusive execution of portions of code known as **critical sections**, i.e., only a single process can be in a critical section at a certain time. Before entering the critical section, a process acquires a lock, causing it to be blocked if the lock has already been acquired by another process. After exiting the critical section, a process releases the lock so that other processes waiting to enter the critical section may proceed [3].

- *barriers* allow two or more processes to synchronize at a certain point in their execution. For example, each process waits at a barrier. When the required number of processes have arrived at the barrier, all of them are allowed to proceed [3].

- *condition variables* are boolean-valued variables that allow processes to wait on logical conditions. If a process waits on a false-valued condition variable, it is blocked. A signal on the condition variable allows one process to be unblocked, while a broadcast causes all processes waiting on the condition variable to be unblocked [3].

- *semaphores* are mechanisms that provide mutual exclusion (as do locks), and synchronization (as do barriers) in a single primitive [4]. Two operations are permitted on semaphores: wait (P) and signal (V). Both are performed atomically.

In some shared memory systems [1][2][5], the memory coherence provided by the system is associated with synchronizing actions made by a program. Thus, in addition to providing mutual exclusion, acquiring and releasing a shared synchronization variable is used to indicate when shared data should be made consistent [7].

1.2 Organization of the Paper

This paper will focus on the implementation of locks, and improves a locking algorithm to be fault tolerant. Section 2 describes the details of the original locking algorithm upon which this work is based. Section 3 describes the modifications to support fault tolerance for the original algorithm and to Raymond's Algorithm. Section 4 provides conclusions.

2 A Distributed Mutual Exclusion Algorithm for Locks

2.1 Terminology

In this paper we adopt terminology used in Quarks since our algorithm is based upon the original non fault-tolerant Quarks algorithm. Each process is called a *node*. A node may or may not run on the same machine as another node. Mutual exclusion is implemented by token-passing between nodes. Before entering a critical section of code, a node must acquire the token, or *lock*. Only one node may possess the lock at particular time, thereby guaranteeing mutual exclusion. After executing the critical section, the node must release the lock. The algorithm from which this work is based assumes that none of the nodes fail.

2.2 Requirements of Distributed Mutual Exclusion Algorithms

Before describing this particular algorithm, it is helpful to remember that mutual exclusion algorithms should be[10]:

- **Free from deadlocks.** Nodes should not wait for messages that will never arrive.
- **Free from starvation.** A node should acquire the lock in a finite time.
- **Fair.** Requests for the lock should be granted in the order they arrive in the system.
- **Fault-tolerant.** The algorithm should be able to continue even if one or more nodes fail.

2.3 Data structures

Each node X keeps track of its participation in the algorithm through several data structures:

- $\texttt{holder}_X$ variable, that tracks the probable holder (node) of the lock. In Figure 1, possession of the lock is denoted by a black dot, and a dashed arrow from node X to node Y means that X's holder variable is set to Y.
- $\texttt{Q}_X$, a queue of requests for the lock
- $\texttt{requested}_X$, a flag that denotes whether X has requested the lock. This allows X to avoid sending another request for the lock.
- $\texttt{held}_X$, a flag that is true if X is using the lock. $\texttt{held}_X$ =true$\Rightarrow\texttt{holder}_X = X$

2.4 Algorithm

Initially, every node's `holder` variable is initialized to a single node (the node that allocated the lock). All queues are initialized to empty, all `requested` and `held` flags to `false`.

When a node X wishes to acquire the lock, a `REQUEST` message is sent for the lock to the node Y named in its holder variable $\texttt{holder}_X$. Y may not be the actual holder, so it forwards the request to the node named in *its* holder variable, and so on. Thus, in the simplest case, a request messages is forwarded along a chain of links ending at the node Z that actually holds the lock, as shown in Figure 1 (b).

1. If $\texttt{held}_Z$ is not set (meaning that Z is not using the lock itself), Z relinquishes the lock, sending it to X.

2. If $\texttt{held}_Z$ is set (meaning that Z is using the lock) the request is added to $\texttt{Q}_Z$. When Z releases the lock, it dequeues the first request in $\texttt{Q}_Z$, and sends the lock, along with the remaining entries in $\texttt{Q}_Z$ to originator of the request.

In both cases Z sets its own holder variable to point to the node where it sent the request. When the lock and queue arrive at the originator node X, the queue is appended to the Q_X. X then acquires the lock.

For example, in Figure 1 (b),

1. node X requests the lock, causing node Y and node W to forward the request (the path of the request messages are shown by solid arrows) to node Z which holds the lock.

2. Then the request is added to $\texttt{Q}_Z$.

3. In Figure 1 (c), node Z releases the lock and sends it to X.

4. The holder variable of Z is reset to X.

The algorithm avoids unnecessary forwarding of messages through the use of the `requested` flags. If a node Y receives a request for the lock from node X, but Y has already requested the lock for itself (denoted by $\texttt{requested}_Y$ being set), then, instead of forwarding the request, it adds the request to its own queue Q_Y. Eventually, Y will receive the lock, acquire it, and release it, and the entries in Q_Y will be serviced.

For example, in Figure 1 (d),

1. node W makes a request, and Z forwards it to the holder X, which adds it to Q_X.

2. This request sets $\texttt{requested}_W$ (denoted in the figure by a black triangle next to W).

3. If node Y now makes a request, as shown in Figure 1 (e), it does not get forwarded all the way to the holder X; instead, the request is added to Q_W, since $\texttt{requested}_W$ is set.

4. In Figure 1 (f), when node X releases the lock, it sends it to node W (the first request in its queue), enabling W to acquire the lock.

5. When W releases the lock, it is sent to the node Y, the first entry in Q_W.

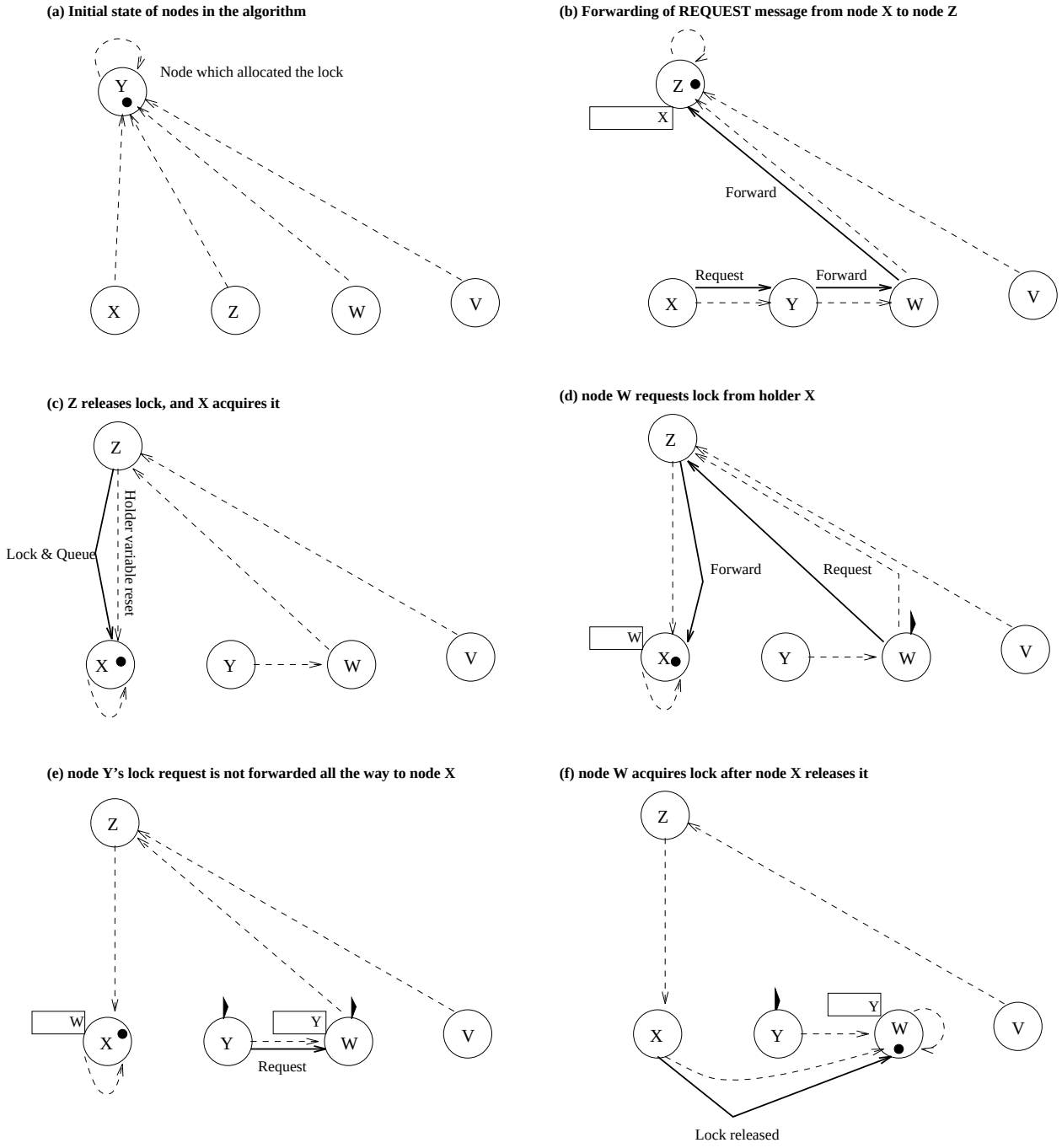


Figure 1: Lock algorithm

2.5 Analysis

Here we analyze the performance of the algorithm described in section 2.4. Initially, the node that allocates the lock is the holder; the holder variables of all other nodes are set to this value. We define *Message complexity* as the number of messages sent to satisfy a request for a lock. The *synchronization delay* is the time between a request for the lock and entry into the critical section[10]. The inaccuracy of a node's holder variable increases the *synchronization delay* experienced by a requests initiated from that node. It also increases the number of messages required to satisfy a request from that node.

The upper bound for the number of messages sent to request the lock is N, where N is number of nodes. This occurs when there is a single chain of holder variables from the requesting node to the holder node, and no node along the chain has itself requested the lock. This consists of $N-1$ messages to forward the request to the holder, and 1 message to send the requester the lock. If all nodes are equally likely to request the privilege, given this linear chain structure, the average distance between the requesting node and the privileged node is $(N+1)/3$, and the number of messages sent is $(N+1)/3+1$. The linear chain structure can occur if nodes request the lock in strict alternation.

The `requested` flag kept at each node mitigates some of the undesirable consequences of inaccurate holder variables. If a node I along the path to the actual holder H has its `requested` flag set, a request is added to I's queue instead of being forwarded to H, thus saving some messages. This modification is possible because I will eventually receive the lock. This saving of messages has a price, however: when the lock arrives at I, it may have an accompanying queue, which needs to be merged with the queue at I, leading to a loss of ordering information, since time-stamps are not maintained with the queue entries. The loss of ordering information means that the algorithm is not fair, in that requests may be served out-of-order. This can be rectified by adding time-stamp information to the requests in the queue and merging queues in order.

2.6 Comparison with Raymond's Algorithm

The mutual exclusion algorithm described earlier is similar to Raymond's algorithm [8]. Raymond's Algorithm assumes a static node configuration arranged in a fixed spanning tree; each node needs to know of the existence of, and communicate with only its immediate neighbors in the spanning tree. The probable location of the lock is determined using holder variables at each node. Raymond's algorithm also uses the equivalent of a `requested` flag to prevent unnecessary request messages being sent. The value of the holder variable at each node is restricted to its neighbors in the spanning tree, unlike the previous algorithm, where the holder variable can be set to any other node in the system.

Raymond's algorithm does not forward requests. If a non-privileged node I receives a request for the lock, it does not forward the request to the node H named by its holder variable. Instead it adds the request to Q_I, and sends a request to H for the lock on behalf of the requester.

For example, in Figure 2,

1. Node B attempts to acquire a lock, entering a request *from itself* in Q_B (Figure 2 (a))
2. It then sends a request message to the holder A. The request is entered into the queue at A.
3. Later, A releases the lock, and it is sent to B.
4. If A (Figure 2 (c)) now receives a request for the lock from node C, it does not forward the request. Instead, it enters Cs request into its own queue, and sends a request for the lock to the holder node B and sets its own `requested` flag.

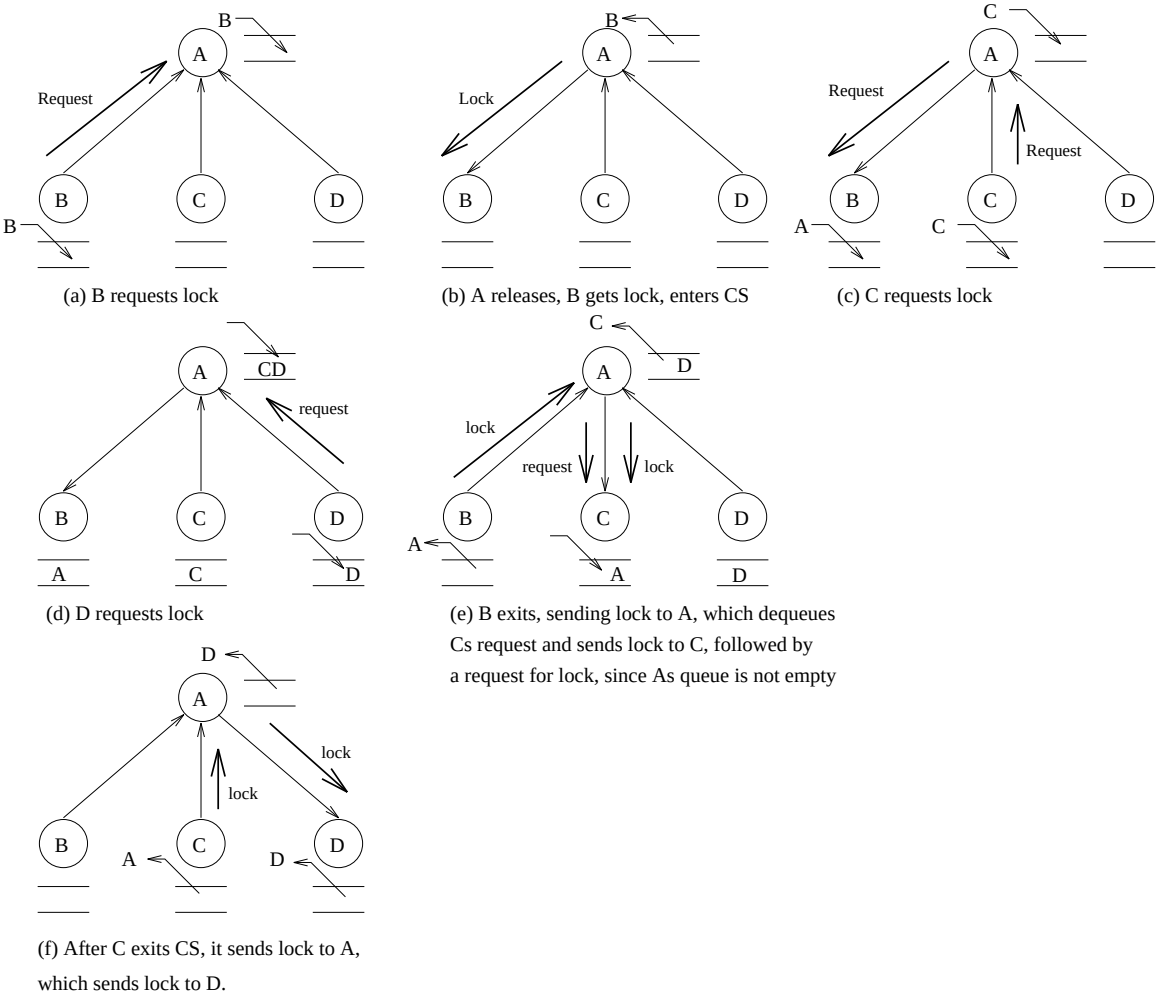


Figure 2: Example Operation of Raymond's Algorithm

5. B sees this as a request from A, and it is entered into Bs queue.

6. If node D now requests the lock (Figure 2 (d)), its request is added to the queue at A, and the `requested` flag prevents the sending of another request message to B.

7. When B releases the lock, it sends just the lock (not the queue, as with the Quarks algorithm) to A, the originator of the first request in its queue.

8. When A receives the lock, it sends the lock to C, the originator of the first request in its queue.

9. Then, if its queue is still not empty, it sends a request for the lock to C.

Raymond's algorithm does not merge queues; however it exhibits unfairness in serving requests closer to the lock holder before requests that are further away. For comparison, the operation of the algorithm described in section 2.4 with the above sequence of requests is shown in Figure 3.

In Raymond's algorithm, the lock must travel through the spanning tree to reach the process which requests it. This inefficiency is not present in the Quarks algorithm, because it attaches the identity of the original requester to the request. When the load is light (i.e., there are few concurrent requests for the lock), the Raymond algorithm will send more messages for each critical section entry, and the lock must retrace the path of the request back to the requester. However, when the load is heavy, two facts reduce the number of messages sent to service each request for the lock:

- Every time a request travels from a requester to the lock holder each node along the path sets its `requested` flag. Therefore, subsequent requests from other requesters do not have to travel all the way to the holder.

- Since requests travel all the way back along the tree to the requester, it is valid to set the requested flags as above. Such a scheme would be incorrect in the Quarks algorithm, since requests in this algorithm are sent back directly to the requester.

In summary, under light loads, the Quarks algorithm requires fewer messages per lock request; under heavy loads, the Raymond algorithm performs better.

3 Adding Fault-tolerance

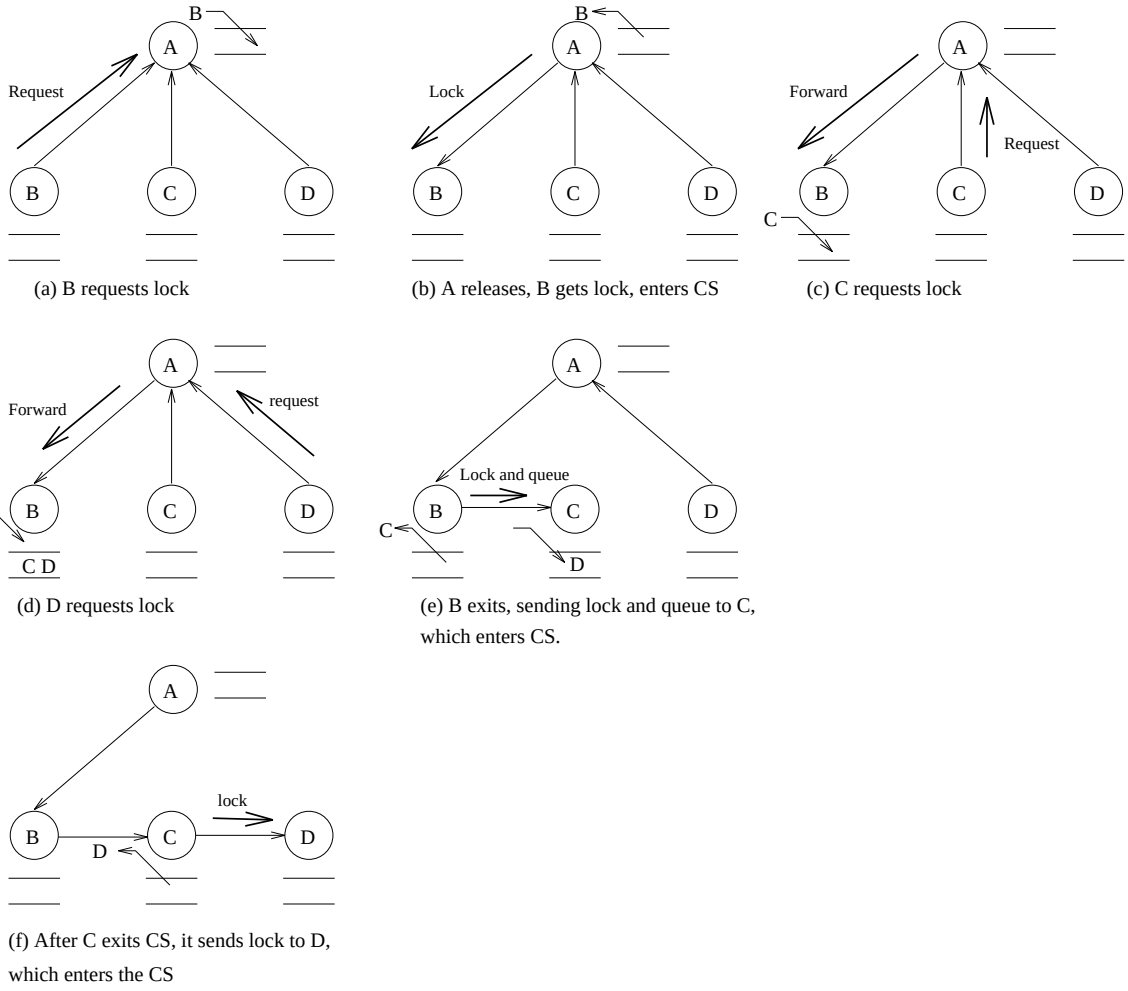
3.1 Consequences of Failure in Original Algorithm

Failure of a node F in the algorithm described in section 2.4, results in the loss of $\texttt{holder}_F$, Q_F, $\texttt{requested}_F$, and $\texttt{held}_F$. The failure of F affects all nodes whose `holder` variables lead to F, either directly or indirectly (through one or more intermediate nodes). All affected nodes will be unable to request the lock. All nodes with requests in Q_F will be indefinitely blocked. In the worst case, F is the holder of the lock, resulting in no node being able to acquire the lock.

3.2 Modifications to Support Fault-tolerance

The algorithm described in the section 2.4 can be made more fault-tolerant by

1. allowing the other nodes to continue when a single node fails, and



(a) B requests lock

(b) A releases, B gets lock, enters CS

(c) C requests lock

(d) D requests lock

(e) B exits, sending lock and queue to C, which enters CS.

(f) After C exits CS, it sends lock to D, which enters the CS

Figure 3: Example of Quarks Algorithm using the same sequence of requests as Figure 2

2. by enabling a failed node to rejoin the algorithm when it restarts.

At each node a number of variables keep track of the state of that node's participation in the algorithm: the `holder` and `requested` variables, and the queue of requests at the node. Fault-tolerance can be supported by recovering the state of a failed node.

Assumptions:

- There are a maximum of N nodes in the system, some of which may be inoperative at a particular time.
- The time t_m required for a message to be transmitted from one node to another is finite.
- During the period of recovery, no other node fails.
- Message delivery is guaranteed.
- Partitioning of the network into one or more disconnected sub-networks does not occur.

3.3 Continuation of Other Sites After Failure

Failure of a single node should not disrupt the algorithm. This can be supported by reconstructing the information stored at the failed node from the state stored at all the other participating nodes. When a node, say node X, fails (see Figure 4 (a) and 4 (b)), the following steps occur:

1. a single node, say node Z, is elected (using a procedure described below) to perform the recovery.
2. Z sends `RECOVER` messages to all other nodes informing them that X has failed, and waits for responses from them (Figure 4 (c)).
3. On receiving a `RECOVER` message, any of the nodes that has its holder variable pointing to X updates it to point to Z.
4. Next, each node sends Z a `STATE` message containing the values of its variables and queue (Figure 4 (d)).
5. Since Z sends a message to all the N-1 nodes other than X, some of which may have failed, it knows when it has received replies from all the nodes that are operational. After Z sends out its messages, it waits for t_{max}, the maximum time to send N-1 `RECOVER` messages and receive `STATE messages` in reply. t_{max} can be estimated by $t_m + t_p + t_m$, where t_m is the time taken to send an `RECOVER` message and receive a `STATE` message in reply, and t_p is the maximum time taken to process a `RECOVER` message.
6. When Z has received `STATE` messages from all other nodes that are operational, it can start recovering the state of the failed node.
   (a) If the `holder` variable of any (there should only be one, at most) of these nodes points to itself, that node is the holder of the lock, and Z updates its own `holder` variable to point to that node. If not, the lock was held by the node that failed; Z elects itself the new holder.

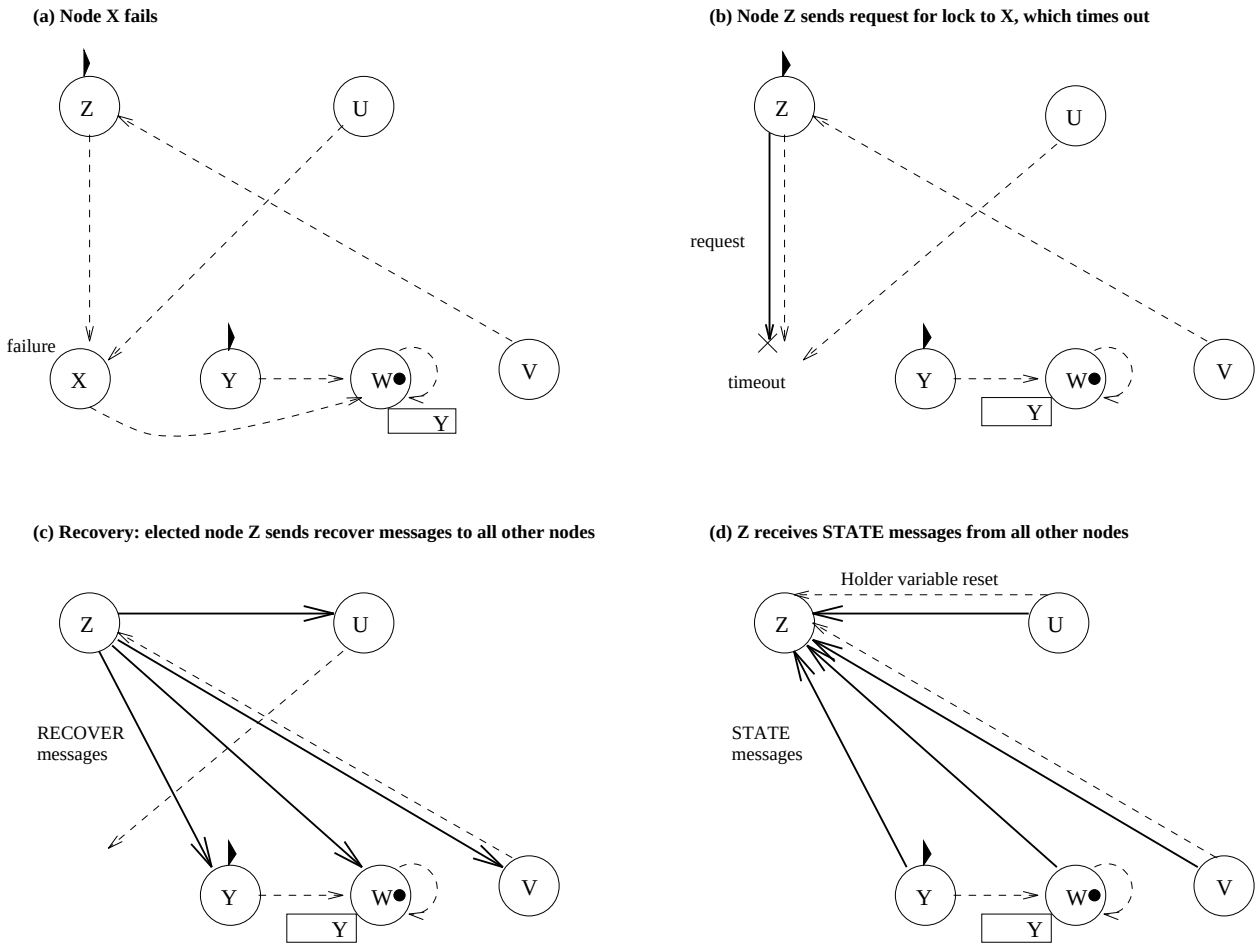


Figure 4: Recovery Procedure

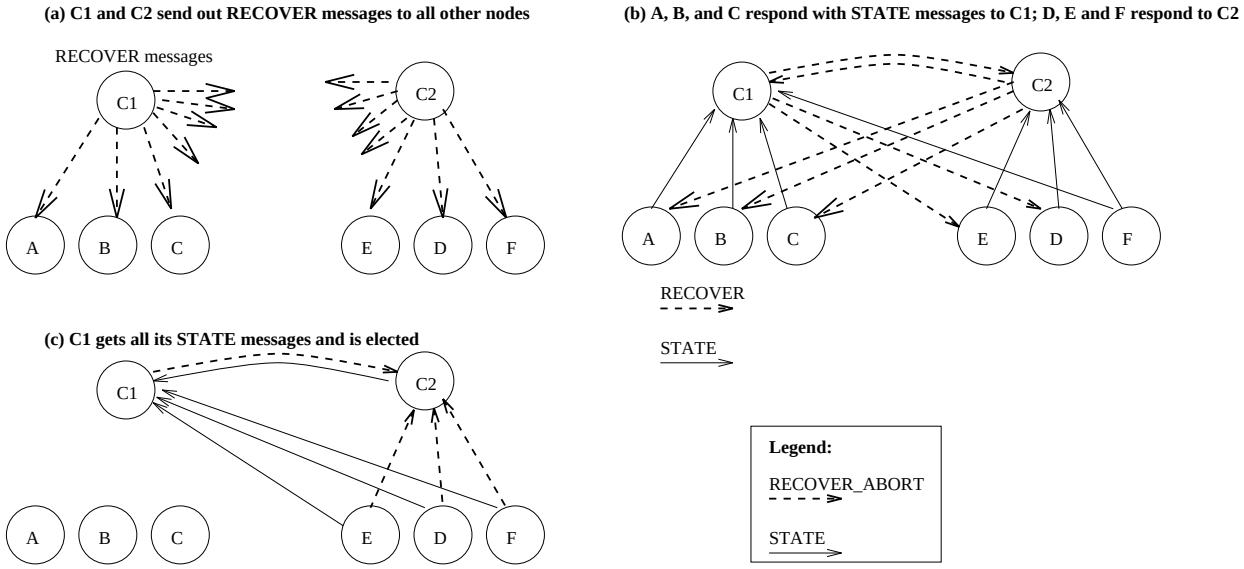


Figure 5: Election of recovery coordinator

(b) All nodes n such that $\texttt{requested}_n$ is true, but $n \notin Q_W$ for all nodes W must have been present in the queue of the failed node. In other words, all nodes that have requested the lock, must have requests present in some queue. If these requests are not present in Z's queue or in the queues of the nodes that sent `STATE` messages to Z, then they must have been present in the queue of the failed node. Thus, requests from these nodes can be added to the queue at Z. The recovery algorithm is able to deduce the entries in the queue of the failed node, but cannot recover their original order.

(c) While recovery is being carried out, the node Z that is elected to carry out the recovery may receive requests for the lock. These requests are accepted and deferred. After recovery is complete, these requests are added to the end of the Q_Z.

3.4 Electing A Node to Carry Out the Recovery

For the above recovery algorithm, a protocol to elect the node to coordinate the recovery is needed. Because the recovery algorithm regenerates lost queue entries, it is essential that only one node carry out the recovery, otherwise duplicate queue entries may be generated.

We considered using the Silberschatz' Bully Algorithm[9] to elect the coordinator, but decided against it because it considers every operating node as a potential coordinator, while our algorithm considers only those nodes that have noticed failure, reducing the number of messages required to resolve conflicts.

Any node that detects the failure of another node (when it times out during a request) may initiate the recovery algorithm. It sends `RECOVER` messages to all other nodes. It may happen that more than one node detects a site failure. Essentially, conflicts are resolved in favor of the node with the lesser node id.

For simplicity, we shall consider only the case where two nodes C_1 and C_2 detect site failure and initiate the recovery. The strategy can be generalized for more than two conflicting nodes.

1. It may happen that C_1's `RECOVER` messages are the first to reach sites A, B and C, while C_2's messages are the first to reach sites D, E, and F (Figure 5 (a)).

2. A, B and C will send C_1 `STATE` messages and if their holder variables point to the failed node, reset their own holder variables to point to C_1.

3. Similarly, C_2 will get `STATE` messages from D, E and F (Figure 5 (b)).

4. Suppose that C_1's node ID is less that that of C_2. When D, E and F get C_1's `RECOVER` messages (after they have already received C_2's messages), they will notice that C_1's node ID gives it precedence, and respond by sending `STATE` messages to C_1, and resetting their own holder variables to C_1 if their holder variables had been set to C_2.

5. When A, B, and C receive C_2's `STATE` messages, they will notice that C_2 does not have precedence over C_1 and respond with a `RECOVER_ABORT` message informing C_2 that C_1 is already performing a recovery and to abort its recovery attempt (Figure 5 (c)).

6. When C_2 receives these messages, it sets its holder variable to C_1. C_2 may also have received a `RECOVER` message from C_1, in which case, it finds out earlier that its recovery attempt is to be aborted, and sends a `STATE` message to C_1. Only the site that receives no `RECOVER_ABORT` messages from any other site is allowed to complete the recovery algorithm.

In the worst case, the site that holds the lock fails, and all the other sites have their holder variables pointing to it. They all notice that it has failed, and initiate the recovery algorithm. This causes $(N-1) \times (N-1)$ `RECOVER` messages to be sent, and $(N-1) \times (N-1)$ `RECOVER_ABORT` and `STATE` messages to be sent in response. Thus, in the worst case, this is an $O(N^2)$, algorithm, similar to the Bully algorithm. However, this worst case is extremely unlikely, requiring both a particular topology, and a simultaneous detection of failure by all sites. In actual practice, only a subset of the sites will detect failure simultaneously, and *only these sites* will be involved in the conflict resolution.

3.5 Failed Node Rejoin

The above recovery algorithm is executed when one node determines that another node has failed using a timeout. If a node, say X, restarts *before* any of the other nodes notices that it crashed, it can use the above algorithm to recover its own state. Essentially, it acts as a node that has has detected its own failure.

1. It sends out `RECOVER` messages to all other nodes.

2. If it receives any `RECOVER_ABORT` messages, it sets its holder variable to the node contained in the `RECOVER_ABORT` message.

3. Otherwise, it initiates a recovery using the `STATE` messages from the other nodes.

   (a) If none of the other nodes have their `holder` variables pointing to themselves, then X must be the holder of the lock; otherwise the node (there should be only one) that has its holder variable set to itself is the holder, and $\texttt{holder}_X$ is set to this node.

(b) $\texttt{requested}_X$ is determined as follows. If, for some node Z, $X \in Q_Z$, then $\texttt{requested}_X$ is `true`. There should be only one node Z. Otherwise, $\texttt{requested}_X$ is `false`.

(c) The entries in Q_X are determined as follows. If $\texttt{requested}_X$ and $\texttt{holder}_X \neq X$, then $\texttt{requested}_X$ is `false`. Otherwise, if, for some node W, $\texttt{requested}_W$ is `true`, but a request from W is not present in the queues of any of the other nodes, W is added to Q_X.

(d) As in the original recovery algorithm, while X is carrying out the recovery algorithm, all requests it receives are deferred by being added to its queue. After the recovery algorithm is complete, the reconstructed requests are placed at the start of the queue. Again, the order of the queue is not recovered. This means that if a site continually fails and rejoins, it is possible that starvation of some nodes could occur

.

3.6 Recovery Preserves Mutual Exclusion

The algorithm in section 2.4 is characterized as a token-passing algorithm because preservation of mutual exclusion depends on the uniqueness of the token, or lock. Assuming reliable message delivery and no failures, the original algorithm, without fault-tolerance added, preserves mutual exclusion, since only one node may possess the lock at any time. If failure occurs, then the node that possesses the lock may have failed, necessitating that the lock be regenerated. Only one lock must be regenerated, in order to preserve mutual exclusion. The node election algorithm above ensures this.

Proof: Suppose (toward a contradiction) that two nodes (the proof can be generalized for more than two nodes) C_1 and C_2 complete the recovery algorithm. This must mean that neither received a `RECOVER_ABORT` message from the other. If C_1's node ID is less than C_2's, this means that C_1 either never sent C_2 a `RECOVER_ABORT` message on discovering its own precedence, which is contrary to the action called for by the algorithm, or that C_2's `RECOVER_ABORT` message never reached C_2, which is not possible, given reliable message delivery. Thus, we have a contradiction.

3.7 Adding Fault-tolerance to Raymond's Algorithm

Raymond's original paper contained a recovery scheme carried out by the failed node when it restarts. However, the case where a node fails, and does not restart for a significant amount of time was not considered. Specifically, in Raymond's Algorithm, each node X knows only of the existence of its neighbors. Each neighbor N of a node leads to a sub-tree, disjoint from the subtrees of its other neighbors. If $\texttt{holder}_X$ is N_1, this means that neighbor N_1 is part of the sub-tree that contains the actual holder node (Figure 6 (a)). If X fails, all nodes in the subtrees that don't contain the holder node will be unable to acquire the lock, since X is their only connection to the sub-tree containing the holder node.

Raymond's algorithm can be made fault-tolerant by having each node maintain information about nodes other than its neighbors, if only for the recovery stage of the process. In particular, each node must know about the neighbors of its neighbors. When a node X fails, one of its neighbors N_1 can recover its state by sending `RECOVER` to X's other neighbors. X's neighbors respond with `STATE` messages, and the state of X can be reconstructed from the state of its neighbors, as shown in Raymond's original paper [8]. The neighbors of X delete X from their neighbor-sets and add N_1 instead, resulting in the structure shown in Figure 6 (b). If all of X's neighbors (including N_1), had their holder variables set to X, then X must have

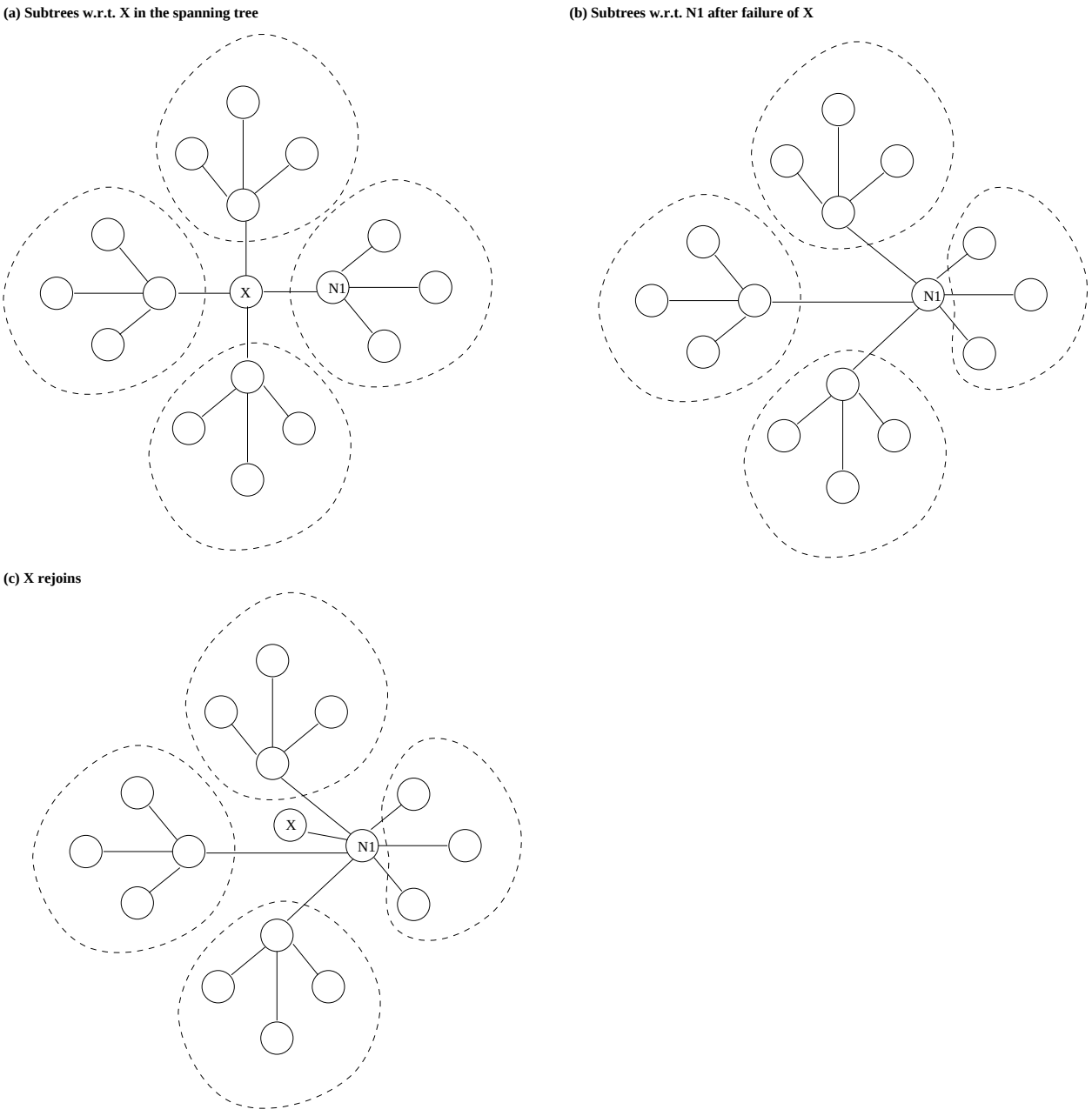


Figure 6: Node failure and rejoin in Raymond's Algorithm

been the holder of the lock; N_1 therefore elects itself the holder. Otherwise, one of X's neighbors (and only one) must not have had X as its holder, meaning that the lock is present in the sub-tree containing this neighbor. Therefore N_1 sets its holder variable to this neighbor.

The case where a failed node X recovers while its neighbors are recovering its state, can be handled as in lock algorithm of section 2.4. When X sends out `RECOVER` messages, its neighbors respond with `RECOVER_ABORT` messages. X chooses one of its neighbors and sends it a message informing it that it is rejoining the tree. This neighbor then adds X to its set of neighbors, resulting in the structure shown in Figure 6 (c).

3.8 Support for Orderly Shutdown

Occasional shutdowns for maintenance purposes, operating system upgrades, or other reasons often require an orderly disconnection to occur. Orderly disconnection allows a node to terminate its participation in the distributed mutual exclusion algorithm in a way that allows the other nodes to proceed without disruption. Before a node D disconnects, it sends `DISCONNECT` messages to all other nodes. If D does not hold the lock, the `DISCONNECT` message contains H, the value of D's `holder` variable. All nodes whose `holder` variables pointed to D now reset their variables to H. If D does hold the lock, the `DISCONNECT` message contains F, the first node in D's queue. In addition, the `DISCONNECT` message to F contains D's queue. Then, if D does not reconnect within a pre-specified time period, or does not reconnect at all, F can continue with the algorithm.

4 Conclusions

The procedures described in this paper add (i) failed node rejoin and (ii) the continuance of other nodes in the event of node failure to the Quarks lock algorithm, an algorithm that makes no provisions for failure. Explicit synchronization mechanisms like locks are more important in shared memory systems like Quarks, because, unlike message-passing systems, processes in shared memory systems do not implicitly synchronize on the sending and receiving of messages. Thus, our improvements are an important step toward making Quarks more robust and reliable and therefore a practical alternative to message-passing systems like PVM.

Our techniques apply equally well to the classic Raymond algorithm, which, in its original form, only supports failed node rejoin. A real system must provide some way for the rest of the nodes to continue in the event of the failure of a single node, since the mean time to repair a failed node is unlikely to be negligible, as assumed by the Raymond Algorithm. The modification to Raymond's algorithm suggested in this paper does not significantly complicate this elegant algorithm, requiring simple changes to each node to keep track of the neighbors of its neighbors, the addition of a continuance procedure, and the modification of Raymond's original rejoin procedure.

References

[1] Bennett, J.K., Carter, J.B., and Zwaenepoel, W., "Munin: Distributed Shared Memory Based on Type-specific Memory Coherence," *SIGPLAN Notices*, vol 25 Mar. 1990, pp. 168-176.

[2] Bershad, B. N., Zekauskas, M. J., and Sawdon, W. A., "The Midway Distributed Shared Memory System," *Proceedings of the 1993 IEEE Spring COMPCON*, 1993, pp. 528-37.

[3] Carter, J. B., Khandekar, D., and Kamb, L., "Distributed Shared Memory: Where We Are and Where We Should Be Headed," *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS-V)*, April 1995, pp. 119-123.

[4] Fleisch, B.D., Hyde, R.L. and Juul, N., "Mirage+, A Kernel Implementation of Distributed Shared Memory on a Network of Personal Computers," *Software-Practice and Experience*, vol 24 Oct. 1994, pp. 887-909.

[5] Lenoski, D., Laudon, J. Gharachorloo, K., Gupta, A., and Hennessy, J., "The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor," *Proceedings of the 17th International Symposium on Computer Architecture*, 1990, pp. 148-59.

[6] Lo, Virginia, "Distributed Shared Memory: A Survey of Issues and Algorithms" IEEE Computer, vol 24 Aug. 1992, pp. 52-60.

[7] Lo, Virginia, "Operating Systems Enhancements for Distributed Shared Memory" Advances in Computers, vol 39, 1994, pp. 199-200.

[8] Raymond, K., "A Tree-Based Algorithm for Distributed Mutual Exclusion," *ACM Transactions on Computer Systems*, vol. 7 Feb. 1989, pp. 61-77.

[9] Silberschatz, A., Peterson, J., and Galvin P., *Operating Systems Concepts*, 4th ed., pp. 595-597.

[10] Singhal, M. and N. Shivaratri, "Advanced Concepts in Operating Systems," McGraw-Hill, New York, 1994, p. 123.

[11] Sunderam, V.S., "PVM: A Framework for Parallel Distributed Computing," *Concurrency: Practice and Experience*, vol. 2 Dec. 1990, pp. 315-339.