Fault Tolerance for Token-based Synchronization Protocols

Frank Mueller *
Humboldt University Berlin, Institut f. Informatik, 10099 Berlin, Germany
e-mail: mueller@informatik.hu-berlin.de

Abstract

The contributions of this paper are twofold. First, a general method to support fault tolerance for token-based synchronization protocols in distributed systems is introduced. Contrary to traditional approaches, this new method uses a decentralized approach to recover from node failures under fail-stop conditions. Neither does it require a centralized manager nor broadcast support.

Second, the method is demonstrated by adopting the protocol for mutual exclusion by Naimi et al. *[21] to tolerate faults. An implementation of the new protocol in a distributed execution environment provides fault detection with fault recovery and false alarm detection. Experiments with different fault injection scenarios underline the feasibility of this new approach. Measurements indicate that our protocol inflicts no overhead in the absence of faults and a factor 5 to 10 overhead in the presence of faults.*

1. Introduction

Executing in a distributed environment requires synchronization to coordinate phases of execution on the nodes, to provide facilities of event notification between nodes and to arbitrate access to common resources through mutual exclusion. These problems can be solved more easily in a shared-memory environment *via* atomic instructions (*e.g.,* test-and-set). In a distributed environment that supports message passing, synchronization has to be provided by protocols through the exchange of messages. Furthermore, single computing nodes in a distributed environment and communication links, *e.g.*, for large clusters, are more likely to fail than processors in a shared-memory multiprocessor (SMP). In a SMP, typically all or none of the processors fail, *i.e.*, the entire SMP node is subject to failure. In this paper, we describe the design of a generic protocol to support fault tolerance for synchronization in a distributed environment.

In distributed environments, synchronization is provided via a series of messages passed between nodes that are participating in the protocol. Several algorithms to provide synchronization for distributed systems have been developed, *e.g.*, to support mutual exclusion [4] or conditional synchronization [18]. They can be distinguished by their approaches as token-based and non-token-based. The former ones may be based on broadcast protocols or they may use logical structures with point-to-point communication. Furthermore, protocols may be centralized or decentralized.

Our support for fault tolerance is applicable to the class of token-based protocols with point-to-point communication along a logical structure. We aim at decentralized protocols to ensure scalability. The main purpose of this protocol is to provide fault tolerance without impacting the synchronization protocol in the absence of failures. In addition, fault detection and recovery should have at most a linear message overhead relative to the number of nodes. A sample implementation of the fault-tolerant protocol is given for mutual exclusion in a distributed execution environment called DSM-Threads [14, 15]. In this work, we assume asynchronous and reliable communication (no loss, duplication or modification of messages) with a bounded message latency. The bounded latency is required to counter the Consensus and the even harder election problem [23]. These problems are addressed in a practice by assuming that a node is faulty if it does not respond within the interval bounded by the latency.

2. Related Work

A number of protocols for distributed synchronization have been published, in particular to address the problem of mutual exclusion in a distributed environment [25, 24, 22, 11, 5, 21]. Chang [4] and Johnson [9] give an overview and compare the performance of such protocols. Several extensions to these protocols have been proposed, *e.g.*, extensions for priority support [8, 3, 16] and for synchronization in distributed shared memory (DSM) systems [18].

Our study focuses on token-based, decentralized protocols for arbitrary synchronization, *e.g.*, mutual exclusion protocols by Raymound [22] and Naimi *et al.* [21]. Chang *et al.* [5] use a dynamic tree similar to Naimi *et al.* [20, 21]. The only difference between the protocols seems to be that

*This work was conducted at the above affiliation. New address: Lawrence Livermore National Lab, Center for Applied Scientific Computing, P.O. Box 808, L-561, Livermore, CA 94551, phone: (925) 424-3642

the root of the tree is piggybacked in the former approach while the latter (and older one) does not use piggybacking. Due to the similarity, we simply refer to the older algorithm in this paper. The above protocols, except Raymond's and Naimi's, have an average message complexity larger than $O(log\ n)$ for a request. But Raymond needs an average of $O(log\ n)$ messages to send the token to a requester while Naimi's only requires one message.

Several fault-tolerant extensions of existing protocols have been proposed. Maekawa's base protocol [11] has been extended to tolerate faults by Bouabdallah and Konig [2]. Mishra and Srimani [13] proposed a centralized protocol and also extended Suzuki's approach [25], which uses broadcasts, for fault tolerance while we offer a decentralized protocol without broadcasts. Agrawal and Abbadi [1] proposed a voting protocol for quorum-based systems that tolerates faults while we focus on non-quorum based protocols. Naimi and Trehel [19] also proposed a fault-tolerant extension to their base algorithm. It encompasses several phases to first detect a failure, then elect a leader to fix the problem before it can be resolved. These phases require multiple broadcasts, which causes a higher message overhead than our approach. Manivannan and Singhal [12] proposed a token regeneration scheme where a site sends a broadcast request if mutual exclusion cannot be granted within a certain period of time. Multiple sites may then offer a token, of which only the most recent one is accepted. This distributed approach handles site and link failures, tolerates network fragmentation and can be applied to Naimi's approach. However, neither do Manivannan and Singhal tolerate false alarms nor do they tolerate tokens in transit (as a message) while our approach handles these cases.

3. Fault Tolerance for Token-Based Synchronization

Synchronization primitives based on a privilege associated with a token provide a simple method to ensure safety properties in a distributed system. For example, in a distributed protocol for mutual exclusion it is assumed that the right to enter the critical section is a property associated with the token. In other words, only the node in possession of the token may enter the critical section. Similarly, conditional synchronization may associate the property of signalling a wait condition with a token, *i.e.*, if a node receives a token, it may wake up the first waiter suspended on the synchronization object.

In a decentralized approach based upon token passing, requests are collected in a distributed queue. New requests are sent either directly to the tail of this queue or will be forwarded along intermediate nodes before reaching the tail and being appended to this queue. During request forwarding, intermediate nodes perform path compression to log the requester as the new tail of the distributed queue, *i.e.*, later requests will be sent directly to this last requester. On the average, path compression results in an overhead of $O(log\ n)$ messages per request for n nodes [21]. These protocols assume point-to-point communication (no broadcast necessary) and do not require any centralized service, which enhances the scalability of the approach.

A fault-tolerant token-based synchronization protocol should preserve the properties of the base protocol, *i.e.*, instead of broadcasts or centralized services, it should rely on point-to-point communication and decentralized control. Several issues have to be addressed in this context:

1. fault detection
2. collection of fault correction information
3. election of a fault handling node
4. fault recovery
5. false alarm recovery

3.1. Fault Detection

In this context, we restrict the discussion to node failure, *i.e.*, fail-stop scenarios of a node. Other failures (transient failures, links failures, etc.) may be handled by the same communication architecture but would require that different fault information be collected and different recovery steps be taken, which are beyond the scope of this paper.

A fail-stop situation is detected by a lack of response. If a node A issues a request to node B and B does not respond, a fault has been detected. In order to prepare for fault recovery, information about the state of each live node has to be collected and combined in a way to exclude the faulty node from any participation in the protocol.

3.2. Collecting Information

The information dispersed throughout the distributed system has to be collected by communication restricted to functioning links. In a broadcast network, a single broadcast request would suffice but, as motivated above, such an approach is subject to limited scalability. Alternatively, a back-up communication structure may be utilized. At first, a ring structure including all nodes is assumed, depicted by the dashed ellipse in Figure 1. The fault detecting node D issues a collection request in both directions along the ring. A receiving node forwards this request on the ring and piggybacks its local protocol state. At some point, a request

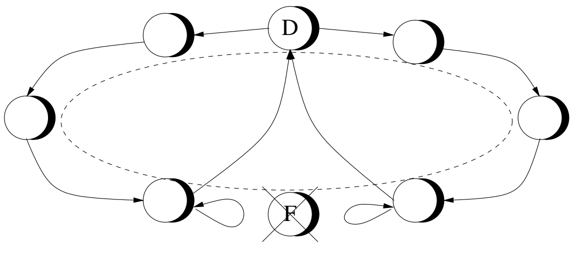

Figure 1. Collection along Ring

is sent to the fault node F, which does not respond. The sending node (neighbor of F on the ring) then forwards the collected fault information to the handling node H. For the sake of simplicity, $D = H$ is assumed here; other cases are discussed in Section 3.3.

A node receiving a collection request becomes aware of an on-going fault handling phase. Certain messages may be deferred and collected in a local queue instead of being sent. In token-based protocols, token-passing messages are subject to deferral during the fault handling phase. Once a node is aware of fault handling, it continues to accept regular synchronization messages but refuses to pass tokens during the fault handling phase. This behavior is motivated by the necessity to prevent a token to be in transit between nodes while the fault handler may regenerate the token, as will be seen in Section 3.4.

The ring also serves another purpose. In the event that fault collection requests circle around the the entire ring, a false alarm is detected since none of the nodes had failed. This will be discussed in more detail in Section 3.5.

The implicit assumption so far restricted the model to a single node failure. To tolerate multiple faults, additional rings with alternate link connections are required. To tolerate a k node failure, each node must be reachable via different rings. The topology of the rings must ensure that each live node can be reached at least by one ring in the event of k arbitrary failures.

Another issue concerns the scalability of the recovery process. Messages circulating around a single ring are subject to latencies with an overhead that grows linearly with the number of nodes n. A more scalable approach is given by a set of rings for each node with $log\ n$ members. The general protocol remains the same but the time overhead would be equivalent to sending $O(log\ n)$ messages in sequence, which is also the average message overhead of the base protocol without failures discussed later.

3.3. Election of a Fault Handler

A node detecting a fault is initially deemed to serve as the fault handler who receives collected fault information, calculates fault recovery actions and initiates them throughout the system. More than one node may, however, detect a node failure at the same time and initiate fault collection. In this case, collection requests will cross each other on the ring. Then, the node with the smallest ID (or some other criteria) is elected the fault handler as depicted in Figure 2.

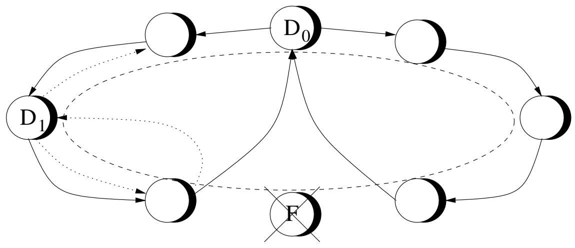

Figure 2. Fault Handler Election

The following design provides this arbitration. Each fault collection request on the ring is acknowledged by a message in the reverse direction of the request. If a second collection request with a different fault detector arrives on a node, the node compares the former and current detector IDs. If the current ID is smaller, the message is acknowledged as usual and passed on along the ring and the current detector becomes the fault handler, as seen for D_0 in Figure 3. Conversely, if the ID is larger, a negative acknowledgement is sent and the request is absorbed, depicted as a *nack* for D_1. This allows the election of a fault handler in the presence of any number of concurrent fault detections.

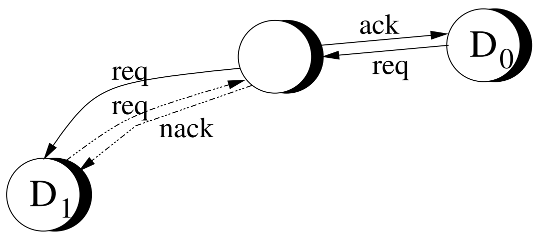

Figure 3. Acknowledgement as Election Support

3.4. Fault Recovery

The recovery process is initiated at the fault handling node once it has received messages from two nodes indicating a node failure. These messages originate from the neighbors of the fault node and contain information collected along distinct portions of the ring. The collected information provides the basis for a reconstruction of a valid protocol state for some synchronization facility in the absence of the fault node. The actual calculation is subject to the protocol details. An example is given in Section 4 for a mutual exclusion protocol.

The calculated recovery state of each node now needs to be distributed to the different nodes. Depending on the protocol, the order in which nodes are reactivated may have an impact on the correctness of the synchronization facility. In particular, distributed queues in the system have to be restored in such a way that requests cannot enter the queue at an arbitrary place. They may only be appended to the tail. In general, distributed queues can be restored by sending recovery requests to each node in the queue in reverse queuing order. Reverse queue reconstruction is the key issue for any fault recovery in token-based protocols.

Figure 4 depicts an example. The tail of the queue (R_2) receives the first recovery message, updates internal protocol state and proceeds to operate just as before the fault detection phase. The other nodes (R_1) follow in this reverse order. If requests from any node reach a recovered

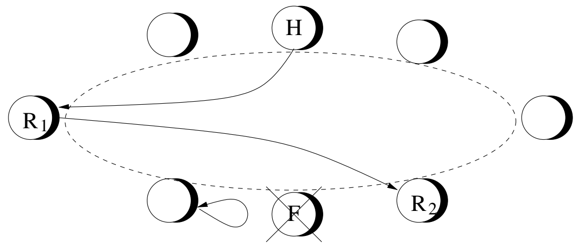

Figure 4. Restoring the Distributed Queue

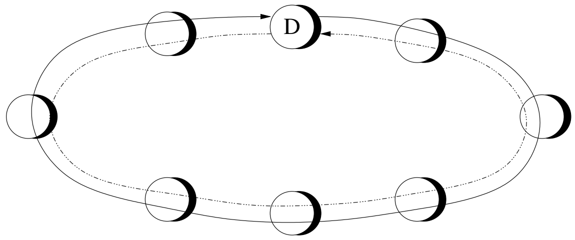

Figure 5. Detection of False Alarms

node, they are simply appended to the distributed queue. If requests reach a non-recovered node, they may be lost under certain circumstances (if forwarded to the fault node). Hence, regeneration of the distributed queue has to take such requests into account.

As mentioned in Section 3.2, token passing is deferred during fault handling. This ensures that the fault handling node uses up-to-date information collected along the ring. The fault handler can assume that no token message is in transit between any two nodes. But it may have to assume that a deferred token message is still present (and has not been reported in the collected information) on some node. Such inconsistencies between collected and actual state in the distributed system have to be taken into account when considering procedures for fault recovery. In the example in Section 4, token requests may continue to circulate during fault detection while the token cannot circulate anymore, for reasons explained earlier.

Fault recovery also includes the repair of the ring structure by excluding the failure node. The neighbors of the failure node on the ring simply skip the failure node. This is achieved by checking if the old neighbor is the fault node and, if so, updating it with a new neighbor node that was included in the recovery message from the fault handler.

3.5. Recovery from False Alarms

The method of the initial fault detection is subject to implementation details and may not be reliable in the sense that a suggested fault may turn out to be caused by longer responses than expected, *e.g.*, due to longer usage of a resource than anticipated. In such a case, the fault detection process may be triggered although there does not exist an actual fault. The described protocol also handles such false alarm. Since collection requests are sent around the ring in both directions and are forwarded by each node without delay, both requests circle around the entire ring back to the handler node, as depicted in Figure 5. When a node on the ring observes two collection messages from opposite directions (initiated by the same fault handler), it can conclude that the ring is intact and may proceed normal operation wrt. the synchronization protocol. Prior to leaving the detection phase, such a node issues all deferred messages as stored in a local queue. The order in which nodes resume operation is irrelevant in this case since there is an absence of faulty nodes. Notice that recovery from false alarm is subject to the assumption that nodes must not fail during the fault detection phase. (If they did, one collection request may circle around the ring while the other is blocked.)

4. Implementation

The general method to support fault tolerance for token-based synchronization of the last section was applied to a specific mutual exclusion protocol in a sample implementation. The underlying base protocol by Naimi *et al.* [21] supports distributed mutual exclusion in a decentralized fashion over a token-based mechanism. This protocol was implemented in DSM-Threads [14, 15], a distributed execution environment supporting distributed shared memory (DSM).

4.1. Token-based Mutex with Path Compression

The protocol by Naimi *et al.* [21] employs a token to indicate the right to enter the critical region of a lock object. Requests that cannot be served right away are registered in a distributed queue originating at the token owner. Once a token becomes available, the owner passes it on to the *next* requester within the distributed queue. In addition, nodes form a logical tree pointing via probable *owner* links towards the root. Initially, the root is the token owner. When requests are issued, they are guided by a chain of probable owners to the tail of the distributed queue. Each node on the propagation path sets its probable owner to the requester, *i.e.*, the tree is modified dynamically. The dynamic adjustment of the tree provides path compression resulting in an average message overhead of $O(log\, n)$ per request for a network of n nodes.

In Figure 6 the root T holds the token for mutual exclusion. The request by A is sent to B following the probable owner (solid arcs). Node B forwards the request along the chain of probable owners to T and sets its probable owner to A. When the request arrives at the root T, the probable owner and a *next* pointer (dotted arc) are set to the requester A. The next request from C is sent to B. B forwards the request to A following the probable owners and sets its probable owner to C. Node A sets its *next* pointer and probable owner to C. When the token holder T returns from its critical section, the token is sent to the sink of the *next* pointer.

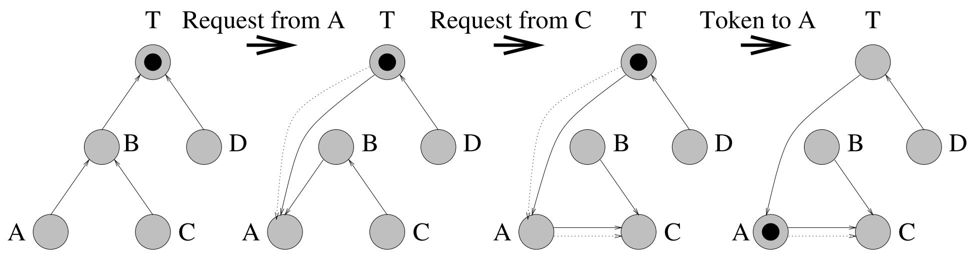

Figure 6. Unprioritized Example

Hence, T passes the token to A and deletes the *next* pointer.

4.2. Fault Tolerance for Distributed Mutex

In the following, implementation details for a fault-tolerant enhancement of the protocol by Naimi *et al.* based on the general outline of Section 3 will be given with respect to fault detection, information collection, handler election, recovery and false alarms.

First, a method for fault detection based on the observation of some malfunctioning system component has to be supplied. Among the possible choices are error notification by the operating system on broken network links and timeout facilities. Our implementation relies on a combination of the two. It is assumed that the protocol for mutual exclusion is utilized by the user to guard short critical sections. The token associated with the critical section is passed between nodes after a short period of execution within the critical section without blocking. We assume that requests should be served within a relative period of $\delta = 100ms$, where δ is required to be larger than the maximum duration of token usage (length of any critical section). Failure to serve such a request results in a timeout causing the requesting node to issue a fault collection request along both ends of the ring. Hence, a timeout results in a suggested fault causing the system to enter the fault detection phase.

The fault collection phase follows the generic protocol described in Section 3. Nodes on the ring are simply informed about the fault detection phase. While the collection request propagates around the ring, no specific information is collected. A faulty node is detected during this phase if a message cannot be sent to a neighbor on the ring. This will be recognized through an error reported by the operating system for a faulty TCP connection or by repeated timeouts on acknowledgements for reliable connectionless communication. [1] When a faulty node is detected, the collection message that propagates along one side of the ring is directly forwarded to the fault handler with an indication of the node that failed, as seen in Figure 1.

In the event of multiple fault detectors, a single fault handler is elected based on the smallest value of the tuple (port number, host name). In DSM-Threads, our execution environment, each node has a single port to receive incoming messages. This port is served by a high priority receiving thread, which ensures that incoming messages are handled right away, regardless of any CPU-bound activity. Since the network typically presents the bottleneck in distributed environments, prompt handling of messages results in increased responsiveness of each node and, thereby, the overall system. The prompt response also supports the detection of false alarms as follows. A node may have initiated the fault detection phase due to a timeout, which may happen if users abuse critical sections for longer lasting operating, thereby slowing the token forwarding sequence. Collection requests are promptly relayed along the entire circle of the ring, regardless of the actions within critical sections. This allows the detection of false alarms in the absence of node failures when both collection requests have propagated around the entire circle.

Fault recovery is driven by the following observations. Any message to forward the token is deferred during fault handling but requests may be issued as always. The fault handler then has to restore a consistent state of the protocol for the overall system. This can be accomplished in two ways. In the first approach, the collected information can be assembled, the faulty node can be removed from the distributed queue and gaps can be bridged before the repair information is sent to the nodes. In the second approach, repair requests are sent to each node but the responsibility to repair internal structures is shifted upon each individual node. The former approach has the advantage that no additional messages are required after the repair request. However, repair instructions based on collected information may be incomplete since the protocol may have advanced past the state reported during the collection phase. Hence, multiple approaches are discussed in the following. The first variant requires that requests are reissued. The second one builds a distributed queue of all nodes. The third one combines these two approaches in a hybrid solution.

4.3. Reissued Requests

The first variant of local action in response to a repair request causes the *owner* to be set to the fault handler while

[1] Low-level communication often implements unreliable send operations where messages may be duplicated or lost, such as given by UDP. We provide a reliable extension to such services by using sequence numbers and requiring acknowledgements for unreliable, connectionless services.

the *next* pointer is cleared. If requests are pending, a new request message is sent following the base protocol, *i.e.*, the message is sent to the probable *owner* and the *owner* is set to the identity of the local node (self). [2] Deferred messages are discarded at this time. When all nodes have been repaired, the token is regenerated in the fault handler and the synchronization proceeds its normal operation. This requires additional request messages in the order of all nodes with pending requests. The message overhead remains linear since all nodes receive a repair request. However, the messages could be sent using a LogP tree structure [6], which would result in only $O(log\ n)$ overhead in time for n nodes.

4.4. Reconstruction of a Distributed Queue

In this variant, a repair request is sent to each functioning node in the reverse order of a distributed request queue, which is being reconstructed. The reverse order is motivated by the observation that synchronization operations may proceed as normal for any node if all following nodes in a distributed wait queue have already been served. The repair request instructs a node to set its probable *owner* to the tail of the distributed queue and the *next* pointer to the next member in the distributed queue (or null). [3] Deferred messages are discarded at this time. When all nodes have been repaired, the token is regenerated in the fault handler and the synchronization proceeds its normal operation, *i.e.*, after execution within a critical section on the fault handler, the token is passed down the distributed queue.

Notice that the queue consists of all nodes after fault recovery, which implies that fault recovery has a linear message overhead in the number of nodes. Furthermore, the token is passed to all nodes due to the decentralized fault repair, even if a node did not issue a requests. This approach comes at the expense of the linear overhead but should result in shorter response times for token reception in the event of faults since requests do not have to be reissued.

4.5. Hybrid Repair

The previous two approaches for repairing can be combined in a hybrid solution. The information gathered during the collection phase is used to reconstruct a distributed queue of pending requests. This queue comprises only those nodes that actually issued requests. It may, however, include already serviced requests and exclude requests that were issued since the collection phase. Hence, a repair message is sent to all nodes, starting with the tail of the reconstructed queue and continuing up to the head, then proceeding with the remaining operational nodes in an arbitrary order. When a node receives a repair message, it always sets the *owner* to the tail of the queue and discards deferred messages. In addition, two cases have to be distinguished. If the node is a member of the reconstructed queue, it sets its *next* pointer as indicated by the fault handler to build up the distributed queue. Conversely, if the node was not a member, it clears the *next* pointer and reissues its requests since they have not been included in the distributed queue yet. The fault handler then regenerates a token as always.

5. Correctness

The fault-tolerant extensions have minimal impact on the base protocol in the absence of failures. Only token messages are deferred during fault collection before false alarm detection. Then, the deferred messages are sent as usual.

Failure of a node causes the system to enter the different phases from fault detection to recovery. First, we have to ensure that the faulty node is detected. We require that each operational node be a member of a ring structure used for fault detection so that messages sent to the fault node from neighbors of the ring will eventually time out. Therefore, a faulty node will be detected.

Second, only one fault handler must be elected, which also ensures that deadlock is prevented during fault handling. If multiple nodes detect a fault simultaneously, only the two messages of the node with the smallest ID will be forwarded around both sides of the ring to the neighbors of the fault node before the messages are sent to this fault detector with the smallest ID. At least one message of other fault detectors will collide with a message with smaller ID, in which case the message with larger ID is absorbed (and not forwarded). Hence, exactly one handler will be elected.

Third, only one token should be present for use in the synchronization protocol, *e.g.*, to ensure mutual exclusion. During fault detection, a token may initially still be used by one node. When this node releases the token, it will be forwarded to a requester either immediately (for pending requests) or when a request arrives. However, token messages are deferred if a node is aware of a fault detector so that no token may be utilized. The neighbor of the fault node with the largest distance from the fault detector is the last node to enter the fault detection phase and may still be utilizing the token. The collection request is then forwarded towards the faulty node, which results in a timeout. After the timeout, the node may no longer utilize the token since timeouts are required to be larger than the maximum duration of token usage. When the node finally forwards the collection request to the detector, the token cannot be in use anymore due to message deferral. After fault recovery, exactly one token is regenerated in the fault handler. Deferred messages (intended to forward tokens) are discarded on all nodes before normal operation resumes. This shows the presence of

[2] In an environment with per-node multi-threading, multiple requests may be issued by different local threads of a node. Then, $k-1$ local requests have to be queued as *next* before the k-th request is sent to the probable *owner*.

[3] In an environment with per-node multi-threading, $k-1$ local requests have to be queued as *next* before the k-th *next* pointer is set to the successor node in the distributed queue.

exactly one usable token during normal operation.

Forth, starvation has to be prevented. A node will eventually receive the token after issuing a request, even if token reception is delayed by fault handling. After fault handling, all nodes participating in synchronization either reissue their requests or receive the token through a reconstructed distributed queue. Hence, token reception is delayed by at most two timeouts (fault detection and send operation from neighbor to faulty node) and n messages for n nodes (around the longer distance of the ring to the faulty node, back to the detector and recovery to the requesting node). If multiple faults occur in sequence (right after repair), this overhead will be incurred repeatedly but the number of operational nodes will be reduced by each fault until the system eventually reaches a stable state (at the latest for a single isolated node). At this stable state, requests will be served according to the base protocol, which prevents starvation through distributed queuing.

Another issue concerns the fairness of requests. The design of the fault-tolerant extensions does not preserve the order in which requests were queued prior to fault detection. Fairness of existing requests could be preserved to some extend by reconstructing the queue based on the collected information, as given by the hybrid variant of repair. As mentioned before, this information is out-of-date and may lead to arbitrary reordering of disconnected portions of the queue. However, either logical timestamps [10] or ordering based on relative fairness [17] could be utilized to provide a partial ordering ensuring fairness as much as possible in a distributed environment. Finally, fragmentation and multiple faults can be tolerated if the ring is reconstructed based on connected fragments of the ring. In case of multiple rings, the probability of fragmentation can be reduced since reachability along any ring is a sufficient condition to include such nodes in a reconstructed ring.

6. Experiments

We performed experiments of an implementation of the described fault-tolerant protocol for mutual exclusion within the execution environment of DSM-Threads [14, 15]. The experiments comprised a set of fault injection scenarios covering failures under different internal protocol states described in the following:

1. A node fails without pending requests.
2. A node fails after issuing a request but before receiving the token.
3. A node fails while holding the token.
4. Communication links are slow.

Case 1 should cause the application to continue without ever detecting a failure. Case 2 causes failure detection and handling such that the current token is eventually discarded and a new token is generated in the handler node. Case 3 also results in failure detection and handling with token regeneration in the handler node but no duplicate token needs to be discarded. Both 2 and 3 cause the ring to be repaired by bridging the fault node. Case 4 causes nodes to enter the failure detection phase but since both collection requests circle the entire ring, a false alarm is detected by each node resulting in sends of deferred token messages. These scenarios do not only provide sufficient empirical test cases for the given protocol, cases 2, 3 and 4 they can also be utilized to evaluate the efficiency of the protocol.

We designed a test environment along the lines of the work by Fu *et. al.* [7] measuring the *Number of Messages per Exchanged critical section entry* (NME) and the response time, *i.e.*, the elapsed time between issuing a token request and the entry to the critical section. The measurements were taken for up to 16 nodes requesting 10 critical section entries each within varying intervals Γ between 5 and 50 milliseconds. The intervals were randomly distributed around Γ within ± 5 milliseconds. If a node releases the lock of a critical section and, due to lack of contention, reacquires it before token forwarding, the measurement is ignored since only remote overhead should be measured.

We used TCP and reliable UDP (with acknowledgements and sequence numbers to prevent losses and duplications) over FastEthernet in a cluster of 16 PCs with 550MHz Athlon processors. The experiments showed correct handling the mutual exclusion property under fault injection for the above scenarios. Our measurements of the response time indicated that the performance of our protocol is not affected in the absence of faults. When a fault occurs, the response time of already issued requests is increased by a factor of 5 to 10, depending on lock contention.

The relative overhead was higher under low contention for the lock since requests were serviced quickly while faults add a constant absolute overhead. For high lock contention, longer response times were observed due to the larger number of simultaneous requests resulting in slower service, which causes the relative overhead to be lower. We did not observe a measurable impact of the number of nodes on the response time during fault handling, *i.e.*, the size of the ring of nodes was not noticeable in the experiments. This is caused by the relatively long timeout periods of $\delta = 100ms$, which dominates the fault overhead (relative to a much lower latency of messages between operational nodes on the ring).

In our experiments, it was sufficient to use a constant timeout period δ. Future experiments may include a dynamically configured δ depending on the number of nodes, the node-to-node latency and, optionally, the level of lock contention. We also plan to study the impact of the choice of δ on false alarms. However, assuming a low probability for node failure, large δ values should be justified to reduce the likelihood of false alarms. The current implementation allows experiments with reconstruction of the distributed queue (see Section 4.4). Current work includes an implementation of reissuing requests (Section 4.3) and the hybrid variant of these two approaches (Section 4.5).

7. Conclusion

In this work, we developed a general method to tolerate faults in token-based synchronization protocols. Particular care is taken to ensure scalability by using a decentralized approach without the necessity of atomic broadcasts. The general design of this method tolerates fail-stop conditions of nodes. An implementation of our fault-tolerant protocol demonstrates its applicability using a distributed token-based mutual exclusion protocol as an example. The protocol implements fault detection, realizes information collection, elects a fault handler and either recovers from faults or detects false alarms. Several variants of fault recovery are discussed. We show the correctness of our protocol. Furthermore, we present experimental results under different fault injection scenarios that underline the feasibility of our approach. In the absence of failures, our approach does not inflict any additional protocol overhead while the overhead in the presence of failures increases response times by a factor of 5 to 10, depending on contention for the token.

Acknowledgement

Claus Wagner contributed the idea of using a ring for detecting faults and suggested to rebuilt distributed queues starting with the tail of a queue.

References

[1] D. Agrawal and A. El Abbadi. An efficient and fault-tolerant solution for distributed mutual exclusion. *ACM Transactions on Computer Systems*, 9(1):1–20, Feb. 1991.

[2] A. Bouabdallah and J.-C. Konig. An improvement of Maekawa's mutual exclusion algorithm to make it fault-tolerant. *Parallel Processing Letters*, 2(2-3):283–290, Sept. 1992.

[3] Y. Chang. Design of mutual exclusion algorithms for real-time distributed systems. *J. Information Science and Engineering*, 10(4):527–548, Dec. 1994.

[4] Y. Chang. A simulation study on distributed mutual exclusion. *J. Parallel Distrib. Comput.*, 33(2):107–121, Mar. 1996.

[5] Y. Chang, M. Singhal, and M. Liu. An improved O(log(n)) mutual exclusion algorithm for distributed processing. In *Int. Conference on Parallel Processing*, volume 3, pages 295–302, 1990.

[6] D. Culler, R. Karp, D. Patterson, A. Sahay, E. Santos, K. Schauser, R. Subramonian, and T. von Eicken. LogP: A practical model of parallel computation. *Communications of the ACM*, 39(11):78–85, Nov. 1996.

[7] S. Fu, N. Tzeng, and Z. Li. Empirical evaluation of distributed mutual exclusion algorithms. In *International Parallel Processing Symposium*, pages 255–259, 1997.

[8] A. Goscinski. Two algorithms for mutual exclusion in real-time distributed computer systems. *J. Parallel Distrib. Comput.*, 9(1):77–82, May 1990.

[9] T. Johnson. A performance comparison of fast distributed mutual exclusion algorithms. In *Proc. 1995 Int. Conf. on Parallel Processing*, pages 258–264, 1995.

[10] L. Lamport. Time, clocks and ordering of events in distributed systems. *Comm. ACM*, 21(7):558–565, June 1978.

[11] M. Maekawa. A sqrt(n) algorithm for mutual exclusion in decentralized systems. *ACM Trans. on Computer Systems*, 3(2):145–159, 1985.

[12] D. Manivannan and M. Singhal. A decentralized token generation scheme for token-based mutual exclusion algorithms. *International Journal of Computer Systems Science and Engineering*, 11(1):45–54, Jan. 1996.

[13] S. Mishra and P. K. Srimani. Fault-tolerant mutual exclusion algorithms. *The Journal of Systems and Software*, 11(2):111–130, Feb. 1990.

[14] F. Mueller. Distributed shared-memory threads: DSM-threads. In *Workshop on Run-Time Systems for Parallel Programming*, pages 31–40, Apr. 1997.

[15] F. Mueller. On the design and implementation of DSM-threads. In *Proc. 1997 International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 315–324, June 1997. (invited).

[16] F. Mueller. Prioritized token-based mutual exclusion for distributed systems. In *International Parallel Processing Symposium*, pages 791–795, 1998.

[17] F. Mueller. Priority inheritance and ceilings for distributed mutual exclusion. In *IEEE Real-Time Systems Symposium*, pages 340–349, Dec. 1999.

[18] F. Mueller. Decentralized synchronization for multi-threaded dsms. In *Workshop on Software Distributed Shared Memory*, May 2000.

[19] M. Naimi and M. Trehel. How to detect a failure and regenerate the token in the $\log(n)$ distributed algorithm for mutual exclusion. In J. van Leeuwen, editor, *Distributed Algorithms, 2nd International Workshop*, volume 312 of *Lecture Notes in Computer Science*, pages 155–166, Amsterdam, The Netherlands, 8–10 July 1987. Springer, 1988.

[20] M. Naimi and M. Trehel. An improvement of the log(n) distributed algorithm for mutual exclusion. In *Int. Conference on Distributed Computing Systems*, 1987.

[21] M. Naimi, M. Trehel, and A. Arnold. A log(N) distributed mutual exclusion algorithm based on path reversal. *J. Parallel Distrib. Comput.*, 34(1):1–13, Apr. 1996.

[22] K. Raymond. A tree-based algorithm for distributed mutual exclusion. *ACM Trans. Comput. Systems*, 7(1):61–77, Feb. 1989.

[23] L. S. Sabel and K. Marzullo. Election vs. consensus in asynchronous systems. Technical Report TR95-1488, Cornell University, Computer Science Department, Feb. 1995.

[24] M. Singhal. A heuristically-aided algorithm for mutual exclusion in distributed systems. *IEEE Trans. Computers*, 38(5):651–662, May 1989.

[25] I. Suzuki and T. Kasami. A distributed mutual exclusion algorithm. *ACM Trans. Comput. Systems*, 18(12):94–101, Dec. 1993.