On Node State Reconstruction for Fault Tolerant Distributed Algorithms

Michael Okun and Amnon Barak
Computer Science Institute,
The Hebrew University of Jerusalem,
Jerusalem 91904, Israel
{mush,amnon}@cs.huji.ac.il

Abstract

One of the main methods for achieving fault tolerance in distributed systems is recovery of the state of failed components. Though generic recovery methods like checkpointing and message logging exist, in many cases the recovery has to be application specific. In this paper we propose a general model for a node state reconstruction after crash failures. In our model the reconstruction operation is defined only by the requirements it fulfills, without referring to the specific application dependent way it is performed. The model provides a framework for formal treatment of algorithm-specific and system-specific recovery procedures. It is used to specify node state reconstruction procedures for several widely used distributed algorithms and systems, as well as to prove their correctness.

Keywords: Distributed algorithms, fault tolerance, state reconstruction, recovery.

1 Introduction

The common formal approach to fault tolerance of distributed algorithms requires a definition of a set of properties that should be fulfilled by the configurations of the distributed system [10]. An algorithm is f-fault tolerant if these execution specifications are not violated, even if up to f failures of certain types occur during the execution. In this paper we consider crash failures of a single node in a distributed system.

For a large family of distributed systems fault tolerance is achieved by recovering the state of a node after it crashes, as opposed to other methods for providing fault tolerance, such as the state machine approach [17] and the primary-backup approach [2], which maintain multiple copies of the same logical component. Several general techniques for recovering crashed nodes, such as checkpointing and message logging methods are used. However, in many cases these methods are not applicable, and the recovery has to be specifically designed for the algorithm. For example, if the algorithm is implemented in hardware (e.g., in a network switch) or in a low software level (e.g., in a distributed operating system) these methods can not be applied for purely technical reasons. Furthermore, when system performance is important, specifically designed recovery has an advantage over general methods. In certain cases this applies even to scientific computations [15], which are the traditional area for checkpointing methods.

A well understood theoretical model of checkpointing and message logging exists, see for example [1, 7]. It is desirable, however, to have a formal approach which deals with node recovery in general, rather than with node recovery via specific methods. The goal of this paper is to provide such a formal model by defining the reconstruction of a node through the requirements it should fulfill (rather than the method by which it is achieved). We use the proposed model to specify reconstruction procedures as well as to prove their correctness, for several widely used distributed algorithms, such as the leader election in a ring and the sequencer algorithms. In addition, the classes of stateless and memoryless algorithms are specially considered. We also study reconstruction at the system level, as opposed to "stand-alone" algorithms, by examining the recovery from crash failure in distributed shared memory systems.

The paper is organized as follows: Section 2 presents a simple model for reconstructibility in synchronous distributed algorithms. Section 3 deals with reconstruction in asynchronous distributed algorithms. In Section 4 we study node reconstruction in release consistent distributed shared memory systems. Our conclusions and some directions for further research are presented in Section 5.

2 Synchronous systems

This section presents a new model for a node state recovery in synchronous distributed algorithms. After describing the system model, we define node reconstructibility and give an example. We then use the reconstructibility property to define stateless and memoryless synchronous algorithms, and describe a recovery algorithm based on the reconstruction method.

2.1 The model

Using the commonly accepted approach, we model each processor (node) N_i in a distributed system as a state machine. A state machine (automaton) has an initial state N_i^{init}, and an action (transition function) that moves the machine from one state to another and sends messages to other nodes.

Computations are performed in rounds. In each round every node uses its current state, its action and the received messages to compute its new state and messages to send. Messages sent to other nodes in the current round are received in the next round. Let N_i^n denote the state of node N_i at the end of round n. Assume that the nodes are connected by communication links. Let $Q_{i,j}$ denote the link (message queue) from N_i to N_j, and $Q_{i,j}^n$ denote the state of the link at the end of round n.

A crash failure [16] during round n causes the failing node to perform no actions after the failure and may cause some of the messages sent by it in round n to be lost. Messages which were sent to a failed node in round $n-1$ and were not processed by it prior to the failure are also lost.

2.2 State reconstruction

Suppose that node N_i fails in round n. Informally, node N_i has reconstruction if the state of the node and the messages it should have sent can be recomputed using information available in states of other nodes. More formally:

DEFINITION 1: A *reconstruction* of node N_i (for round n) is a special action R, that transforms N_i^{init} into N_i^n, i.e., the state node N_i would have had at the end of round n if it had not failed. In addition, R must send all the messages N_i would have sent during round n in a normal (non-failed) execution. If termination is allowed, i.e., a node may finish its execution at some round n, reconstruction exists only for the rounds in which the node is active.

DEFINITION 2: Let Π be some set of nodes so that $i \notin \Pi$. Node N_i *has reconstruction with respect to* Π if provided with $\{N_j^n\}_{j\in\Pi}$ and$\{Q_{j,i}^n\}_{j\in\Pi}$ there exists a reconstruction for the node for round n.

2.2.1 Example: Leader election

Consider the following leader election algorithm in a ring, where each node has a unique identifier. In the first round each node sends its identifier to its left neighbor. In the following rounds each node compares the identifiers it receives from its right neighbor to its own and forwards the received identifier (to the left neighbor) only if its value is higher. If a node receives its own identifier it declares itself as a leader and sends a *halt* message to its left neighbor. Each node which receives a *halt* message forwards it and terminates at the end of the round. A node which receives the *halt* message and is not a leader (yet) declares itself a non-leader.

To support reconstruction with respect to its neighbors, a node should execute the above algorithm and also remember the message it sent in the previous round. This algorithm is shown in Figure 1, where m_2 and m_1 are the messages sent by the node in the previous and the current rounds, respectively.

Constants:
$myID \in \mathbb{N}$
Variables:
$leader \in \{true, false, null\}$, initially $null$
m, m_1, m_2 - messages, initially $\perp$

1: $m_2 := m_1$; $m_1 := null$
2: **if** $m = \perp$ **then** /*1st round*/
3: $m_1 := myID$
4: Let m be the message received from the right neighbor, or $null$ if no message was received
5: **if** $m = halt$ **then**
6: $m_1 := halt$
7: **if** $leader = null$ **then**
8: $leader := false$
9: terminate at the end of this round
10: **else if** $m = myID$ **then**
11: $m_1 := halt$; $leader := true$
12: **else if** $m > myID$ **then**
13: $m_1 := m$
14: **if** $m_1 \neq null$ **then**
15: send m_1 to left neighbor

Figure 1: The leader election algorithm

1: /*N_{i-1} is the right neighbor,N_{i+1} is the left one*/
2: **if** $N_{i+1}.m_2 = \perp$ **then** /*failure in 1st round*/
3: $m_1 := myID$; $m := null$
4: **else**
5: $m_2 := N_{i+1}.m$; $m := N_{i-1}.m_2$
6: **if** N_{i+1} terminated $\vee$ $N_{i+1}.m = halt$ **then**
7: $leader := true$
8: **if** $m = halt$ **then**
9: $m_1 := halt$
10: **if** $leader = null$ **then**
11: $leader := false$
12: terminate at the end of this round
13: **else if** $m = myID$ **then**
14: $m_1 := halt$; $leader := true$
15: **else if** $m > myID$ **then**
16: $m_1 := m$
17: **if** $m_1 \neq null$ **then**
18: send m_1 to left neighbor

Figure 2: Leader election node reconstruction procedure

Figure 2 presents the reconstruction procedure for a node N_i using the states of its neighbors. Assume that the failure is not in the first round (which is dealt with in a special way), then m_2 is copied from m of the left neighbor. Similarly, the message received in this round (m) is copied from m_2 of the right neighbor. If the left neighbor of the node terminated or terminates in the current round, the node is the leader because non-leader nodes terminate before their left neighbor. Finally, it is necessary to process the message m.

2.3 Memoryless and stateless synchronous algorithms

The above reconstruction of a leader election algorithm is an application of the sender-based message logging [7] method. In the leader election algorithm (Figure 1) the message to be sent is independent of the current state of the node (the value of *leader* variable), and is determined solely by the message received (the first round is an exception). Thus, the availability of the received message on the sender node reduces the reconstruction problem to that of state reconstruction. More generally, the following can be defined:

DEFINITION 3: A node is *k-round memoryless* if the messages it received in rounds $n, ..., n-(k-1)$ and the messages it sent in rounds $n-1, ..., n-k$, allow reconstruction of the messages to be sent in round n.

CLAIM 1: If N_i is a k-round memoryless node and it has a reconstruction of its state only (without the requirement to resend any messages) respective to the other nodes, then the algorithm can be transformed into an algorithm in which the node has reconstruction.

PROOF: We augment the original state of the nodes with: (i) the messages received in rounds $n, ..., n-(k-1)$, i.e., the last k rounds; and (ii) messages sent in rounds $n, ..., n-k$. The reconstruction for N_i is as follows: (i) according to the assumption, the "original" part of the state N_i^n can be reconstructed; (ii) messages received in rounds $n, ..., n-(k-1)$ are copied from the appropriate sender; (iii) messages sent to other nodes in rounds $n-1, ..., n-k$ are copied from their receivers; (iv) since the node is k-round memoryless the messages which should be sent in round n can be computed using the message history, which is now available. □

Intuitively, a node is k-round memoryless if the messages sent and received in rounds $n-1, ..., n-(k-1)$ allow to compute the part of N_i^{n-1} which is required to respond to the messages received in round n. If the state

N_i^{n-1} is completely determined by these messages the node is stateless. More formally:

DEFINITION 4: A node N_i is *k-round stateless* if the messages it sent and received in rounds $n-1, ..., n-k$ allow reconstruction of N_i^{n-1}.

Note that from the definition it immediately follows that k-round stateless node is $(k+1)$-round memoryless.

CLAIM 2: If a node is k-round stateless then the algorithm can be transformed into an algorithm in which the node has reconstruction.

PROOF: Similar to the proof of claim 1. □

2.3.1 Example: Matrix multiplication

From the previous discussion it follows that stateless algorithms have simple and efficient reconstruction procedures. It is interesting to note that some well known algorithms can be easily transformed into stateless ones, e.g., Cannon's parallel algorithm for matrix multiplication (see for example [5]). In this algorithm it is convenient to give the nodes identifiers of the form (i, j), where $0 \leq i, j < p$. The two square matrices A, B to be multiplied and the result matrix C are similarly divided into blocks $\{A(i,j), B(i,j), C(i,j)\}_{0 \leq i,j < p}$. In round r $(0 \leq r < p)$ node $N_{i,j}$ computes $C(i,j) := C(i,j) + A(i, j+i+r \bmod p) \cdot B(j+i+r \bmod p, j)$. In the end of the round each node sends the block of A and the block of B that it held to the nodes which need them in the next round. Note that in the original algorithm the reconstruction of $N_{i,j}$ in round r involves the computation of $\sum_{k=0}^{r} A(i, j+i+k \bmod p) \cdot B(j+i+k \bmod p, j)$.

To transform the algorithm into a 0-round stateless, one needs to send the C blocks (instead of the A or the B blocks). More specifically, in round r, node $N_{i,j}$ computes $C(i+j+r \bmod p, j) := C(i+j+r \bmod p, j) + A(i+j+r \bmod p, i+2j \bmod p) \cdot B(i+2j \bmod p, j)$. The modification preserves the message complexity and the communication pattern of the original algorithm. Since the modified algorithm is 0-round stateless the sender-based message logging method reduces the reconstruction to repeating the computation of the current round.

2.4 Recovery

Reconstruction is a method for recovering a system from node crash failures. The following claim provides an example of reconstruction-based recovery protocol in a totally connected system.

CLAIM 3: If in a synchronous distributed algorithm for a subset of nodes Σ there exists a reconstruction with respect to all other nodes, then at the expense of a slowdown by a constant factor a crashed node from Σ can be transparently recovered.

PROOF: The original algorithm is executed in every fourth round (the other three are used for transparent recovery in case of failure). Suppose that in round n the original algorithm is executed. The nodes are required to store the messages sent (until the $n+2$ round). In round $n+1$ each node in Σ sends a *keep-alive* message to all other nodes. These messages are received in round $n+2$, they are ignored unless a message from some node in Σ was not received. In this case each node sends its state to the node whose keep-alive message is missing. In addition, the messages which were sent to the node in round n are resent. In round $n+3$ the failed node uses the reconstruction action to compute the state it would have had at the end of round n and to send the messages it would have sent. A node may accept such a message (in round $n+4$) only if it received no message from the failing node in the $n+1$ round. In the $n+4$ round the next round of the original algorithm is performed with the crashed node in the correct state and with all previous messages correctly delivered. □

We note that the algorithm described above is suitable only for transient crash failures in which the failure lasts for at most 3 rounds, i.e., a node which fails in round n is back online in round $n+3$. If the crash is permanent a *backup* node should be used instead of the crashed one.

3 Asynchronous systems

This section deals with node reconstruction in asynchronous distributed algorithms. We present the model, then define node reconstruction and give an example of a Sequencer algorithm.

3.1 The model

As in the synchronous case, we use state machines to model nodes (processors) in asynchronous systems. The automaton used to represent a node can have multiple actions of two types: (*i*) internal actions; and (*ii*) actions triggered by external events, e.g., *receive* events. Unlike the synchronous case, there are no rounds. Instead, each automaton (node) performs its actions (tasks) independently of the others nodes. The internal actions may have preconditions, i.e., they may be executed only when the state of the automaton satisfies such conditions. Furthermore, we assume the executions are *fair*, i.e., each enabled task is eventually executed.

During each state transition a message may be sent to any other node. It is possible to send a different message to any number of nodes in the same transition. Communication is modeled by two links (one in each direction) that hold messages between any pair of communicating nodes. Let $Q_{i,j}$ denote the link (message queue) from node N_i to node N_j. In the model, messages can not be duplicated but may be lost. A $messageLost(m, N_j)$ event is delivered by the $Q_{i,j}$ queue to the sending node (N_i) when it loses message m. The automaton *must* have a *receive* action for every incoming link and *messageLost* action for every outgoing link.

The state of the whole system is described by the states of the nodes and by the contents of the queues between all communicating pairs of nodes. An execution (a run) is a sequence $\pi_0\pi_1\pi_2\ldots$ of actions performed by the nodes. We use a discrete time, at a time $t \in \mathbb{N}$ the π_t action is performed. Let N_i^t be the state of node N_i at time t, and $Q_{i,j}^t$ be the state of $Q_{i,j}$ at that time.

3.2 State reconstruction

We now define the reconstruction action R for node N_i. Suppose that a node crashes at some time t_{fail}, during the execution. Following the synchronous case, we would like to define reconstruction as an action which at some later time ($t_{rec} > t_{fail}$) computes $N_i^{t_{fail}}$ and resends the messages sent by N_i, which were lost after it failed. However, if the last message to be accepted was sent by N_i at time t_{last}, it is sufficient to require from the reconstruction to compute a state which is reachable from $N_i^{t_{last}}$, because no other node has any knowledge about the transitions of N_i in the $[t_{last}, t_{fail}]$ time interval. We note that this last fact is exploited in message logging algorithms for asynchronous systems [12].

The formal definition of reconstruction is as follows:

DEFINITION 5: A *reconstruction* for node N_i is a special action R such that if executed at $t_{rec} > t_{fail}$, R transforms N_i^{init} into a state which could have been reached from $N_i^{t_{last}}$ by a sequence of *receive* actions for incoming messages which N_i received during the $[t_{last}, t_{fail}]$ interval, *messageLost* actions for messages in transition after t_{last}, and internal actions. In addition, R is required to send the messages sent by the described sequence of actions. If termination is allowed, i.e., a node may finish its execution at some time t, reconstruction exists only for the time in which the node is active.

Next, we define reconstruction with respect to a set of nodes Π ($i \notin \Pi$). After N_i's failure, for each $j \in \Pi$, at some time $t_j > t_{fail}$ the queue $Q_{i,j}$ is empty. Let t_{rec} be any time such that $\forall j \in \Pi\ t_{rec} > t_j$.

DEFINITION 6: Node N_i *has reconstruction with respect to* Π if provided with $\{N_j^{t_j}\}_{j\in\Pi}$ and $\{Q_{j,i}^{t_j}\}_{j\in\Pi}$, there exists a reconstruction for the node.

The ability to lose messages is a crucial part of the definition of reconstruction for the asynchronous model. It allows the algorithms to specify what changes can be made to the messages in the message queues *after* they were sent. Such changes allow the reconstruction to send messages different from those sent prior to failure. Below we give an example to demonstrate the definition and clarify this last point.

3.2.1 Example: Sequencer

A Sequencer [18] is a server that allocates (natural) numbers to its clients so that: (*i*) each number is allocated at most one time; (*ii*) each client gets the numbers in a strictly increasing order; and (*iii*) all numbers are used, i.e., (k was allocated) $\Rightarrow$ ($\forall i < k$, i will be eventually allocated).

Sequencers are required in many distributed applications, e.g., for timestamps, for transaction serial numbers in databases and in broadcast algorithms. Formally, a distributed system consists of a Sequencer automaton, a number of client automata and communication links between the Sequencer and each client.

Variables:
$counter \in \mathbb{N}$, initially 0
$last[1..\#clients]$ - array of integers,
initially all entries are -1
$lostMessages$ - list of (message, nodeID) pairs,
initially empty ($null$)
On $receive(m, N)$
1: $last[N] := m$
2: $counter++$; send $counter$ to N
On $messageLost(m, N)$
3: append (m, N) to $lostMessages$
On $sendLost$ /*internal action*/
Require: $length(lostMessages) > 0$
4: **if** $length(lostMessages) = 1$ **then**
5: send m to N, where $(m, N) \in lostMessages$
6: $lostMessages := null$
7: **else**
8: let $(m_1, N_1), (m_2, N_2)$ be the first two messages in $lostMessages$
9: **if** $last[N_1] < m_2 \wedge last[N_2] < m_1$ **then**
10: send m_2 to N_1; send m_1 to N_2
11: **else**
12: send m_1 to N_1; send m_2 to N_2
13: $lostMessages := lostMessages \setminus \{(m_1, N_1), (m_2, N_2)\}$

Figure 3: The Sequencer algorithm

Suppose that a Sequencer has 3 clients N_1, N_2, N_3. After it received a request from each client (in increasing order), it sent 1 to N_1, 2 to N_2 and 3 to N_3. Suppose that the Sequencer crashed and only N_2 received its message, i.e., N_2 got 2, while N_1 and N_3 issued requests which were handled by the Sequencer but were lost (otherwise they would still be in $Q_{i,Sequencer}(i = 1, 3)$). From the above scenario, it is impossible to know which number was sent by the Sequencer to which client, any feasible reconstruction may as well send 3 to N_1 and 1 to N_3. To make such reconstruction valid the specification of a Sequencer must allow this reordering of assigned numbers.

Figure 3 shows one possible specification. The Sequencer holds array ($last$) of the last sequence number which was already received by each client (this is achieved by the clients sending their last number in order to request a new one). In addition to the *receive* and *messageLost* actions the Sequencer has an internal action *sendLost* (lines 4 - 13) which retransmits lost messages. If more than one message is lost the *sendLost* action makes reassignments of the sequence numbers in a way which does not violate condition (ii), above.

Variables:
$first, last, prev \in \mathbb{Z}$, initially -1
$requested \in \{true, false\}$, initially $false$
On $request$
1: **if** $requested = false$ **then**
2: send $last$; $requested := true$
On $messageLost(m)$
3: send m
On $receive(m)$
4: $prev := last$; $requested := false$
5: **if** $m \neq last + 1$ **then** /*new interval*/
6: $first := m$
7: $last := m$

Figure 4: The Sequencer client algorithm

Consider now the clients of the Sequencer. Even in the case of a system with two clients if the only information available in the states of the clients is the last allocated number, there exists no reconstructible algorithm for which conditions (i)-(iii) hold (we omit the formal proof of this statement). A possible modification in this case (the one with minimal cost from the complexity point of view) is to hold the bounds of the last interval of accepted sequence numbers. In addition, the one before the last sequence number should be remembered, in order to enable the correct reconstruction of the $last$ array of the Sequencer. The algorithm for the client is presented in Figure 4, where $[first, last]$ is the last interval of sequence numbers continuously assigned to the client and $prev$ is the sequence number assigned to the client before $last$. We note that *request* is an external action that is activated by some upper application layer.

The reconstruction algorithm is presented in Figure 5. In the first stage, the $last$ array is reconstructed. Then the unused sequence numbers are reassigned in a way that does not violate condition (ii) for nodes whose incoming message was lost.

CLAIM 4: The algorithm in Figure 5 is a correct reconstruction of the Sequencer with respect to its clients.

PROOF: We need to show an execution continuing from t_{last} that produces the same state and messages

Variables:
$N', m \in \mathbb{Z}$, initially -1
C, M - sets of integers, initially $\emptyset$

1: **for** $i := 1$ to $\#clients$ **do**
2: **if** $N_i.requested \wedge Q_{i,Sequencer} = \{\}$ **then**
3: $last[N_i] := N_i.last$
4: **else**
5: $last[N_i] := N_i.prev$
6: $m := min_{1 \leq i \leq \#clients}\{N_i.first\}$
7: $counter := max_{1 \leq i \leq \#clients}\{N_i.last\}$
8: Let M be the set of missed sequence numbers
9: $C := \{N | N.requested \wedge Q_{N,Sequencer} = \{\}\}$
/*set of clients whose incoming message was lost*/
10: **while** $M \neq \emptyset$ **do**
11: $m := min(M)$; $M := M \setminus \{m\}$
12: Let $N' \in C$ be the client with
$last[N'] = min_{N \in C} last[N]$
13: $C := C \setminus \{N'\}$; send m to N'
14: **while** $C \neq \emptyset$ **do**
15: Let $N' \in C$ be a client with minimal ID
16: $C := C \setminus \{N'\}$
17: $counter++$; send $counter$ to N'

Figure 5: Sequencer reconstruction procedure

as the reconstruction algorithm. We construct this execution in three phases. The first phase is the execution in the $[t_{last}, t_{fail}]$ interval, exactly as it was performed before the failure. The second phase consists of $\lceil length(lostMessages)/2 \rceil$ *sendLost* actions, resulting in an empty *lostMessages* list. Note that the *last* array after these two phases is equal to the one reconstructed in Figure 5, lines 1-5: for client N, that has a message in its incoming queue, $last[N]$ is $N.last$, while for client that has no incoming message, $last[N]$ is $N.prev$.

Consider the set M of sequence numbers (line 8). At the end of the second phase these numbers are in transmission, however not necessarily to the same clients as in the reconstruction. The third phase of the execution consists of a sequence of *messageLost* and *sendLost* actions that result in an assignment equal to the one produced by the reconstruction procedure. The sequence is constructed in the following way: suppose that the reconstruction sends $m := min(M)$ to N_1, while at the end of the second phase m is sent to N_2, and m' to N_1. Since $m < m'$ and $last[N_1] < last[N_2]$, the loss of m and m' followed by *sendLost* results in m being sent to N_1 (as in the reconstruction) and m' to N_2. For any new number m generated (on line 17), the $m > last[N_i]$ $(1 \leq i \leq \#clients)$ inequalities hold, thus any difference between the current assignment and that of the reconstruction action can be eliminated in a similar way. □

We note that the Sequencer algorithm in Figure 3 should be regarded as a *specification*, the actual implementation can be different. For example, a sequencer that simply resends a lost message is a correct implementation, since it produces only executions allowed by the specification. Furthermore, the reconstruction procedure in Figure 5 is suitable for the implementation described above. This implementation, however, completely obscures the fact that even after m was sent to N_i, it is still possible for another number to be assigned instead of m. Thus, for purposes of formal reasoning about the system we should use the specification for which the existence of reconstruction was shown.

3.3 Recovery

In a distributed asynchronous system there is no way to detect a crash failure of a node because a "slow" node can not be distinguished from a crashed one. Several models which allow to circumvent this difficulty were proposed, e.g., failure detectors [4] and models that incorporate time [3, 13]. In these strengthened models crashes are detectable, thus recovery algorithms similar to the one presented for the synchronous systems can be designed. Below, we give a brief description of recovery by reconstruction in such strengthened asynchronous model. As in the synchronous case, we consider a distributed algorithm with a subset of nodes Σ for which there exists a reconstruction with respect to all the other nodes, and a transparent recovery after a failure of single node from Σ should be performed.

When a node N_i detects that some node $N_c \in \Sigma$ crashed, N_i sends its state to N_c (and resends it if the message containing the state is lost). Besides that the execution of the original algorithm continues as if no failure happened, the only exception is that any messages which were sent by N_c prior to failure are ignored, as if $Q_{c,i}$ lost them after N_c's crash.

When N_c receives the state messages from all other nodes it performs the reconstruction. The reconstruction

action should assume that the $\{Q_{i,c}\}_{i \neq c}$ message queues are in an empty state, if at some later stage $Q_{i,c}$ delivers a message which was sent before the message with N_i's state it should be ignored. In order to allow N_c distinguish between messages which N_i sent before it learned about N_c's failure and messages sent after that, N_i must mark the later messages as such. Similarly, N_c must have special mark on the messages it sends after recovery, so that other nodes can distinguish them from messages sent by N_c before the failure. Additional point which should be noticed is that N_c may receive regular messages, i.e., messages of the original algorithm, concurrently with the state messages. Since messages may be lost it is even possible that a regular message from a node is received before its state message. These regular messages should be stored till the completion of the reconstruction and then processed as usual.

Finally we note that the above algorithm may be applied if the failure of N_c is transient, i.e., at some time after the crash the node will be back online. If the crash is permanent a backup node can be used.

4 System level fault tolerance

Distributed data management systems, such as Distributed Shared Memory (DSM) or distributed file systems have a management layer responsible to guarantee synchronized access to shared data from different nodes as well as correct management of data directories. Fault tolerance of the corresponding layers is crucial for the correct operation and the integrity of the whole system.

In this section we use our formal model to investigate reconstructibility of various DSM systems. We begin by presenting reconstruction of the star mutual exclusion algorithm, which serves as a representative of the algorithms used by system management layer. Then we show how the reconstructibility of the memory on a failed node depends on the consistency model and the data propagation algorithms used by the DSM system.

4.1 Star mutual exclusion

Star mutual exclusion [14] is an example of asynchronous distributed algorithm for mutual exclusion in totally connected network. Its performance is ranked among the best [9] in the class of *token* based algorithms.

In the star mutual exclusion algorithm the waiting queue is distributed among the nodes, where each node has a variable (*next*), pointing to the next node in the queue. A special *root* node holds a variable (*last*) pointing to the last node in the distributed queue of the nodes waiting for the token. To add itself to the queue a node sends a request message to the root. The root node forwards the request to the last node in the queue, then the requesting node becomes the new tail node.

Variables:
$last$ - nodeID, initially ID of the node with the token
$nodeOrder$ - linked list of nodeIDs, initially holds $last$
$lostMessages$ - list of (message, nodeID) pairs, initially empty ($null$)

On $receive(req, N)$
1: send $req(N)$ to $last$; $last := N$
2: remove N and nodes in front of it from $nodeOrder$
3: append N to the tail of $nodeOrder$

On $messageLost(req(N_2), N_1)$
4: append $(req(N_2), N_1)$ to $lostMessages$

On $sendLost$ /*internal action*/
Require: $length(lostMessages) > 0$
5: **if** $length(lostMessages) = 1$ **then**
6: send $req(N_2)$ to N_1, where $(req(N_2), N_1) \in lostMessages$
7: $lostMessages := null$
8: **else**
9: let $(req(N_2), N_1)$ and $(req(N_4), N_3)$ be the first two messages in $lostMessages$, s.t. N_1 precedes N_4 in $nodeOrder$
10: send $req(N_4)$ to N_1; send $req(N_2)$ to $last$
11: $last := N_3$
12: $lostMessages := lostMessages \setminus \{(req(N_2), N_1), (req(N_4), N_3)\}$
13: modify $nodeOrder$ s.t. N_4 follows right after N_1 and N_2 follows after the tail node, making N_3 the new tail of $nodeOrder$

Figure 6: The Star mutual exclusion algorithm for the root node

A specification of the algorithm executed by the root node is presented in Figure 6. Note that a loss of

outgoing messages by the root node may cause a change of order in the waiting queue (lines 9-13). Such behavior is required in order to allow reconstruction (a similar specification method was used for the Sequencer in Section 3).

To reconstruct the root node, the order of the waiting nodes in the distributed linked list is restored using their *next* pointers. If a message sent by the root to some requesting client was lost after a crash, the *next* pointer of the client remains *null*. Hence, messages lost after the root node crashed leave the distributed queue divided into several fragments. To complete the reconstruction, messages which reestablish an order among the fragments (link them together) are sent, and the fragment that contains the node with the token is placed at the end of the list. In order to correctly reconstruct the *nodeOrder* linked list each node must remember the node from which it received the token for the last time. A formal proof of the correctness of the above procedure is similar to that of the Sequencer (claim 4).

4.2 Reconstructibility of Release Consistent Distributed Shared Memory

In release consistent DSM systems [6], local memory updates are propagated only at synchronization points. Synchronization is performed by distributed mutual exclusion algorithm, e.g., the star mutual exclusion. There are two classes of protocols for transferring the modifications to other nodes. The *update* protocols send the modifications to other nodes, while the *invalidation* protocols send only a short invalidation message of modified pages, and invalidated pages are retrieved only when accessed. In both classes the *multiple-writer* method is used, i.e., each node propagates only the part of the page which it actually changed (*diff*), thus allowing multiple nodes to perform concurrent writes to the same page. To compute the diffs, each node keeps the original content of the pages it modified after the last synchronization point. The availability of the original pages is important for reconstruction of the memory of a failed node.

If an invalidation protocol is used, reconstruction is obviously impossible. For example, a node that crashes right after the synchronization point, was observed by other nodes in a state that can not be reconstructed, since the updates are present only on the failed node. Thus, a protocol which guarantees that the updates are propagated to additional nodes must be used, see for example [8] for a class of such update protocols for sequentially consistent DSM. In what follows, we assume that an update protocol is used and discuss how the consistency model affects the reconstructibility of the nodes in DSM system.

4.2.1 DSM with Eager Release Consistency (ERC)

In the ERC protocol a node is required to deliver its page modifications to all the other nodes holding the page prior to any lock release operation. Thus, to guarantee that any modification is propagated to additional nodes, each page should be held by at least two nodes. In this case, reconstruction respective to all other nodes is possible because at any stage after a crash each page can be retrieved in a consistent state from any node holding a copy of that page. Observe that such a retrieval is possible because local modifications produced after the last synchronization point are kept separately. After all the pages held by the node prior to its crash are copied, its state is the same as the state the node would have had if it did not fail but rather produced no local updates during the $[t_{last}, t_{rec}]$ interval, where t_{last} is the time of the last synchronization just before failing, and t_{rec} is the recovery time.

4.2.2 DSM with Lazy Release Consistency (LRC)

In the LRC protocol a node which acquires a lock must receive all required page modifications prior to the acquire operation. The modifications that the acquirer has to obtain are determined according to the timestamp vectors (used to represent the happened-before-1 partial order) of the releaser and the acquirer. Thus, to guarantee that any modification is propagated to additional nodes, each modified page of the releaser should also be held by the acquirer.

We show that the above specification of LRC is not reconstructible. Consider a three node system in which node N_1 updates page p and sends the update to node N_2. Later N_2 updates p and sends it to N_3, then N_1 crashes. The definition of reconstruction requires to recover N_1 to a state it could have reached from the state it

had after sending the update to N_2 (the last message sent by N_1). Since in the described scenario no messages are sent to N_1 (unlike ERC), the state N_1 had after sending its update to N_2 should be reconstructed. However, this is impossible because the content of p is no longer available: the page available on nodes N_2 and N_3 already includes the updates of N_2.

From the last example it follows that in order to allow reconstruction it should be possible (though not obligatory) for a releaser to propagate its modifications to nodes other than the acquirer. This behavior can be represented in our formal model if, in addition to regular messages, it is allowed to send messages for which *messageLost* event is not delivered, even if the message is lost. The releaser uses such messages to send its modifications to nodes other than the acquirer. Note that since these additional messages are not acknowledged, they may be obsolete at the time of their arrival. This specification change allows to reconstruct the node by retrieving pages and diffs from other nodes.

4.2.3 DSM with Scope Consistency (ScC)

ScC [11] is a further relaxation of the released memory consistency model. In this model each lock is associated with a scope. A lock acquire opens the scope associated with the lock and lock release closes its scope. An update is performed in the context of all the scopes that are open at the updated time. In the ScC protocol, prior to the acquire operation, a node which acquires the lock must receive only the modifications performed in the scope of the lock.

Nodes of a DSM system which uses ScC protocol are not reconstructible. Moreover, there are no specification changes (similar to the one presented for LRC) which enable reconstruction. The reason is the inherent ability of nodes to "inform" others only about some parts of the modifications. Consider for example the case in which node N_1 updates page p_1 in scope S_1, and page p_2 in scope S_2. If afterwards S_2 is opened on another node, the page p_2 is sent to this node, but p_1 is not. Now suppose that N_1 fails. It is impossible to reconstruct N_1 because the state of page p_1 was not propagated (yet) to any other node.

5 Conclusion and future work

This paper presents a general formal approach to state reconstruction of a crashed node in both synchronous and asynchronous environments. The model is used to present formal reconstruction procedures for the leader election, matrix multiplication, sequencer and the star mutual exclusion algorithms, and to analyze reconstructibility in release consistent distributed shared memory systems. In addition, we defined the classes of stateless and memoryless algorithms and described corresponding message logging schemes. For practical purposes, the main contribution of this paper is the asynchronous model considered in Section 3.

The work described in this paper can be extended in several directions. First, we note that while our definition of reconstruction is for "pure" asynchronous algorithms, a similar definition can be given for algorithms which use (perfect) failure detectors, e.g., a modification of the star mutual exclusion, in which the client nodes are allowed to crash and the root node uses a failure detector to remove the faulty clients from the waiting queue. In order to define reconstruction in this model, the failure detector should be regarded as a source of nondeterministic inputs, similarly to the incoming messages.

The definitions of reconstruction considered in this paper required to recover the state and the messages of a single node without interfering with the state of any other node, which is the approach used by the pessimistic message logging techniques. In some algorithms it can be reasonable for the recovery procedure of node N_i to modify states of other nodes in the system, e.g., as in a checkpoint-based recovery. A related issue is the concurrent failure of several nodes and their subsequent recovery. It is desirable to investigate the reconstruction in these new settings.

Finally, we note that although all the algorithms considered in this paper required no stable storage for recovery purposes, this is not the general case. An algorithm which uses a stable storage can use our model by representing the storage devices as additional nodes that do not crash, or as a part of a node state which survives crash failures, similarly to the node identifier.

Acknowledgments

We would like to thank Robbert van Renesse for his valuable comments in the early stages of this research.

This research was supported in part by the Ministry of Science, the Ministry of Defense and by a grant from Dr. and Mrs. Silverston, Cambridge, UK.

References

[1] L. Alvisi and K. Marzullo. Message Logging: Pessimistic, Optimistic, Causal and Optimal. *IEEE Trans. on Software Engineering (TSE)*, 24(2), pp. 149-159, February 1998.

[2] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg. The Primary-Backup Approach. Ch. 8 in *Distributed Systems* edited by S. Mullender, Addison-Wesley, 1993.

[3] F. Cristian and C. Fetzer. The Timed Asynchronous Distributed System Model. *IEEE Trans. on Parallel and Distributed Systems (TPDS)*, 10(6), pp. 642-657, June 1999.

[4] T. D. Chandra and S. Toueg. Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM (JACM)*, 43(2), pp. 225-267, March 1996.

[5] J. Demmel, M. Heath, and H. van der Vorst. Parallel Numerical Linear Algebra. *Acta Numerica* 2, pp. 111-198, 1993.

[6] S. Dwarkadas, P. J. Keleher, A. L. Cox, and W. Zwaenepoel. Evaluation of Release Consistent Software Distributed Shared Memory on Emerging Network Technology. *Proc. 20th Annual Int. Symposium on Computer Architecture (ISCA)*, pp. 144-155, San Diego, CA, May 1993.

[7] E. N. Elnozahy, L. Alvisi, Y.M. Wang, and D. B. Johnson. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. To appear in *ACM Computing Surveys*.

[8] B. D. Fleish, H. Michel, S. K. Shah, and O. E. Theel. Fault Tolerance and Configurability in DSM Coherence Protocols, *IEEE Concurrency*, 8(2), pp. 10-21, June 2000.

[9] S. S. Fu, N. Tzeng, and J. Chang. Empirical Evaluation of Mutual Exclusion Algorithms for Distributed Systems. *Journal of Parallel and Distributed Computing (JPDC)*, 60(7), pp. 785-806, July 2000.

[10] F. C. Gartner. Fundamentals of Fault-Tolerant Distributed Computing in Asynchronous Environments. *ACM Computing Surveys*, 31(1), pp. 1-26, March 1999.

[11] L. Iftode, J. P. Singh, and K. Li. Scope Consistency: A Bridge between Release Consistency and Entry Consistency. *Theory of Computing Systems*, 31(4), pp. 451-473, August 1998.

[12] P. Jalote. Fault tolerant processes. *Distributed Computing*, 3(4), pp. 187-195, 1989.

[13] N.A. Lynch. Distributed Algorithms. Morgan Kaufmann, 1996.

[14] M. L. Neilsen and M. Mizuno. A dag-based algorithm for distributed mutual exclusion. *Proc. 11th Int. Conf. Distributed Computing Systems (ICDCS)*, pp. 354-360, Arlington, TX, May 1991.

[15] J. S. Plank, Y. Kim, and J. Dongarra. Fault-Tolerant Matrix Operations for Networks of Workstations Using Diskless Checkpointing. *Journal of Parallel and Distributed Computing (JPDC)*, 43(2), pp. 125-138, June 1997.

[16] F.B. Schneider. What Good are Models and What Models are Good? Ch. 2 in *Distributed Systems* edited by S. Mullender, Addison-Wesley, 1993.

[17] F.B. Schneider. Implementing Fault-tolerant Services using the State Machine Approach: A Tutorial. *ACM Computing Surveys*, 22(4), pp. 299-319, December 1990.

[18] A. S. Tanenbaum and M. van Steen. Distributed Systems Principles and Paradigms. Prentice Hall, 2002.