Fault Tolerant Dynamic Agent Systems

by

James M Roewe

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2005

Author ....[illegible]..
Department of Electrical Engineering and Computer Science
February 16, 2005

Certified by...[illegible]....
Larry Rudolph
Principal Research Scientist
Thesis Supervisor

Accepted by [illegible]..............................
Arthur Smith
Chairman, Department Committee on Graduate Students

Fault Tolerant Dynamic Agent Systems

by

James M Roewe

Submitted to the Department of Electrical Engineering and Computer Science
on February 16, 2005, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Partial system snapshots reduce the cost per node to only depend on the size of the connected group instead of the size of the full system. These groups can be determined during system operation by using the communication patterns between nodes. The number of nodes that must rollback after a failure is limited to the size of these snapshot groups, reducing the work lost. These changes to snapshot algorithms are necessary because the cost per node for a snapshot increases and the expected time between failures decreases as the size of the system grows.

Thesis Supervisor: Larry Rudolph
Title: Principal Research Scientist

Acknowledgments

I would like to express my appreciation to Professor Larry Rudolph for his support in starting my research at NTT, discussing ideas about the research, and making comments on this thesis.

I would like to thank my supervisor at NTT, Tadashi Araragi, for starting me in research on fault tolerance. He helped me understand the system he had worked on and possible extensions to it.

I would also like to thank Adam Oliner who gave me a better idea of the details and issues of real world distributed systems.

I am also grateful for all of the support from my family and my friends at MIT.

Contents

List of Figures

Chapter 1

Introduction

Fault tolerance for distributed systems is more reliable and efficient when it adapts to the nodes' communication pattern. Many algorithms save the application state in checkpoints and restore this state after a failure, reducing the amount of work lost. Algorithms that create a set of consistent checkpoints for the entire system are less efficient because of the overhead of coordinating all of the nodes in the system. On the other hand, uncoordinated algorithms that create new checkpoints independently at each node have less overhead, but can't guarantee that there is a set of checkpoints that form a consistent global state. Because many applications have long periods of sparse communication, new checkpoints should be coordinated among small groups of connected nodes, reducing the overhead of organization while still producing a global saved state.

1.1 Distributed System Model

The distributed systems discussed in this thesis consist of many connected nodes. Each has a processor, temporary memory, connections to other nodes, and stable storage (Figure 1-1). Temporary memory is typically some form of RAM or processor cache. When the node fails or resets, all data in the temporary memory is lost. Data written to stable storage survives across failures, but writing to stable storage is much slower than writing to memory. Each node may have its own unique hard disk or

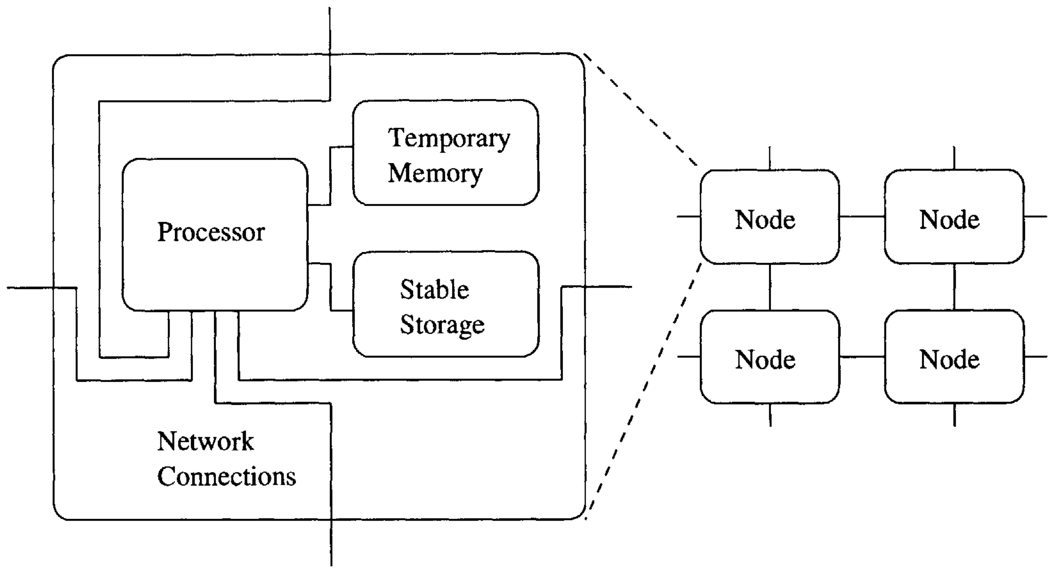


Figure 1-1: Node Layout.

may have a partition on a shared stable storage. Nodes interact with each other by sending messages over the network. A set of physically connected nodes is referred to as a system. A job or application is a set of interacting nodes working together for a common task.

1.2 Failure

Failures in a distributed system can range from RAM going bad to a loose cable to an air conditioning unit failing and causing a set of nodes to overheat. When one of these failures occurs, the affected nodes may have their memory corrupted or may completely go down. They will be fixed, swapping out any bad components for spares and then restarted to allow the system to continue functioning as quickly as possible. In addition, other nodes that had depended on state in these nodes will become corrupted and will need be restarted. If a failure occurs in a distributed system without fault tolerance, the entire system must restart from the beginning and all of the work that had been completed prior to the failure is lost.

A common model of failures in a distributed system is to assign each node a MTBF (Mean Time Between Failures) [4]. The MTBF takes into account all types of failures that can occur in a node and models them as a Poisson process with MTBF as the arrival rate. The calculation of MTBF typically assumes that failures at different nodes are independent. However, failures in a system are often correlated, e.g. a shared power source for a set of nodes may be interrupted, causing all of them to fail at the same time.

In a system of N nodes, the MTBF of the system drops to $MTBF/N$ because a failure at any one node affects the entire system. If the MTBF of a single node is 1,000 hours, a system of 65,000 nodes would have a MTBF of less than a minute. The MTBF of the system will actually be better than this because some of these failures will occur around the same time, only resulting in a single reset of the system. However, the MTBF will decrease as the number of nodes grows. A job that requires longer than the MTBF of the system will have a low chance of finishing.

1.3 Existing Checkpoint Systems

Checkpoint algorithms save nodes' volatile data to permanent storage in a checkpoint so that it survives failures [3]. After a failure, nodes restore the most recent copy of this data and resume operation. Ideally, all nodes would save their state at the same time creating a saved image of the system at that instant. This approach is not possible because nodes are not synchronized. Their internal clocks drift from when they were started and they can't be resynchronized because communication between them is delayed by the network.

While nodes can't synchronize the creation of checkpoints, they can organize when they create checkpoints to form a consistent global state of the system. This approach is called coordinated checkpointing. The basic procedure for a coordinated checkpoint in MPI terms is shown in Figure 1-2. Each node waits until all nodes have reached a convenient time in the application to save their state. When every node has successfully saved its state to permanent storage, they replace the previous checkpoint with the new checkpoint and continue the application. With this algorithm, failure recovery still simply requires restoring the last committed checkpoint. This direct

Checkpoint Procedure

1. Barrier Sync
2. Copy volatile data to a checkpoint in permanent storage
3. Barrier Sync
4. Commit checkpoint to replace previous checkpoint
5. Barrier Sync
6. Continue the application

Figure 1-2: Coordinated Checkpoint Procedure in MPI terms.

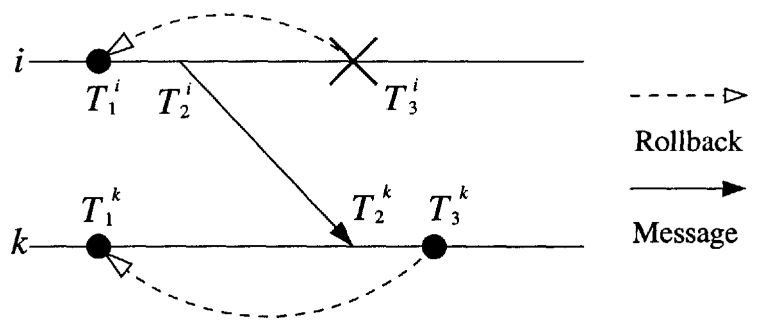


Figure 1-3: Node i fails causing Node k to rollback to T_1^k.

approach is very inefficient because it requires the entire application to wait three times for all nodes to reach the same point and send notifications.

An alternative to coordinating checkpoint creation is to let nodes checkpoint independently and only coordinate nodes when they need to rollback. When choosing a set of checkpoints to restore, if one node rolls back to before having sent a message, then the recipient of that message must rollback to before having received it. In a failure-free system, it would not be possible for a node to have received a message that had not been sent (Figure 1-3). Node i saved its state at T_1^i and sent a message to k at T_2^i. Node k saved its state once at time T_1^k, received the message from i at T_2^k, and saved its state again at time T_3^k. When i fails at T_3^i, it rolls back to its checkpoint at T_1^i. Node k must roll back to T_1^k instead of T_3^k because i rolled back before sending the message that k received.

1.3.1 Uncoordinated Checkpointing

Uncoordinated checkpointing [5, 9, 8] has a low overhead for creating checkpoints because nodes save their state without waiting for other nodes. These algorithms work best for systems that have a lower failure rate and will be creating checkpoints much more often than rolling the system back. Instead of creating a single set of consistent checkpoints, nodes store multiple checkpoints and choose the best later. In exchange for the low overhead of creating checkpoints during failure free operation, there is a high cost during recovery, complex garbage collection algorithms are needed, and there is a possibility of the "Domino Effect" rolling the system all the way back to its initial state.

After a failure, nodes must choose a set of checkpoints to restore. These checkpoints must meet the condition described above that no node can receive a message that has not been sent. Nodes choose the checkpoints according to message tracking information that is saved with the checkpoints. The algorithm must either collect all of this information at one node to make one completely informed decision about which checkpoints to use or must iteratively rollback one node at a time until they are all consistent. The application can't continue running until this recovery algorithm has completed.

Checkpoints that do not form part of a consistent global state or do not form the most recent consistent state can, in theory, be discarded. In a long running system, these checkpoints must be removed to make space for new checkpoints. A garbage collection algorithm determines the earliest set of checkpoints that could be needed, called the recovery line. The algorithm doesn't stop the application from running, but it does create overhead in the system from communicating with all other nodes to find the recovery line.

The most significant problem with uncoordinated checkpointing is that it may be necessary to restore nodes to their initial state. Depending on the timing of message passing and when the checkpoints were saved, rolling back nodes may cause each other to continue rolling back to an earlier checkpoint until they have reached the

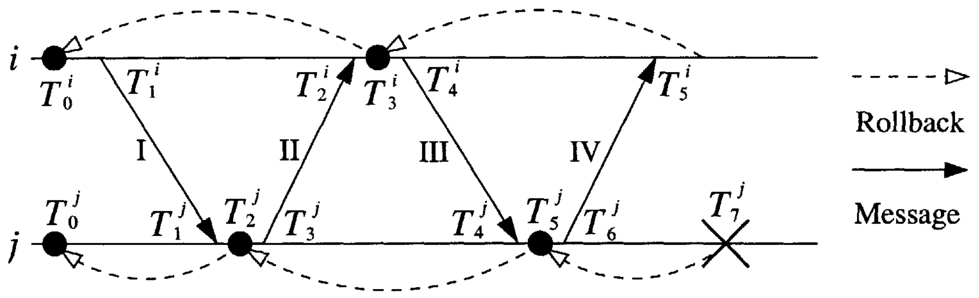


Figure 1-4: Node j fails, causing a rollback which leads to the Domino Effect.

starting state of the system [9]. Figure 1-4 shows the domino effect. At T_7^j, j fails and must roll back to its last checkpoint at T_5^j. The rollback will undo $Msg_{j \rightarrow i}$ IV, causing i to roll back to T_3^i. Similarly, i's rollback will undo $Msg_{i \rightarrow j}$ III, which forces j to rollback further and undo another message. This process will repeat until i and j have rolled back to T_0^i and T_0^j respectively.

1.3.2 Coordinated Checkpointing

Coordinated checkpoint algorithms trade the need for garbage collection algorithms and the possibility of the domino effect for a higher overhead during checkpoint creation. Nodes cooperate to save their state in a single consistent set of checkpoints that will replace the previous set of checkpoints. If failures are more frequent, then guaranteeing that the work is saved is worth the higher overhead. Each node only needs to save one finalized checkpoint and one tentative checkpoint [6]. The finalized checkpoint is the previous completed checkpoint that will be restored if a failure occurs. The tentative checkpoint is stored during the snapshot process until it is finalized and replaces the previous checkpoint. If the tentative checkpoint is unable to finish successfully, then the new checkpoint is discarded and the old checkpoint is still used in the case of a failure.

Chandy and Lamport [2] developed an efficient system for saving the global state of a system in a FIFO (First In First Out) network. In their algorithm, the global state of the system is saved without forcing the application to wait for all of the nodes to save their state. A node begins a snapshot by saving its state to permanent

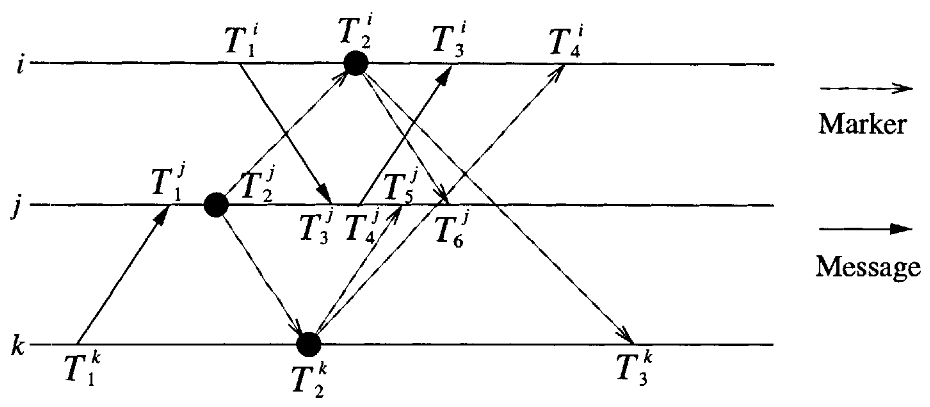


Figure 1-5: Chandy Lamport Algorithm.

storage and then sending a special message called a snapshot marker to every other node. When a node receives a marker, if it has not already joined the snapshot, it saves its memory and sends markers to all other nodes. After joining a snapshot, each node saves all messages from nodes until it receives a marker from them. All messages sent by a node before it joins the snapshot will arrive before the marker and all messages sent after joining the snapshot will arrive after the marker because the network is FIFO. When a node has received a marker from every node in the system, it completes the checkpoint and marks it as the new permanent checkpoint. Each node is responsible for saving its state and all messages in the inbound network link that were sent before the sender saved its state.

In Figure 1-5, j saves its state at T_2^j and sends markers to nodes i and k. At T_2^i and T_2^k, i and j, respectively, save their state and send markers to the other two nodes. Node j saves $Msg_{i \to j}$ which arrives at T_3^j with the checkpoint. At T_6^j, j completes the checkpoint because it has received markers from both i and k. Nodes i and k similarly complete the checkpoint when they receive markers from the other nodes. $Msg_{k \to j}$ is not saved directly, but k's state is saved after the message was sent and j's state is saved after having received the message. $Msg_{i \to j}$ was sent before the snapshot began, but received after j had already saved its state. This message is saved with the checkpoint so that it can be replayed if the system rolls back to this checkpoint. In terms of the global state, the message appears as if it was in the network when all of the nodes saved their state. $Msg_{j \to i}$ is not saved in j's state because it is sent

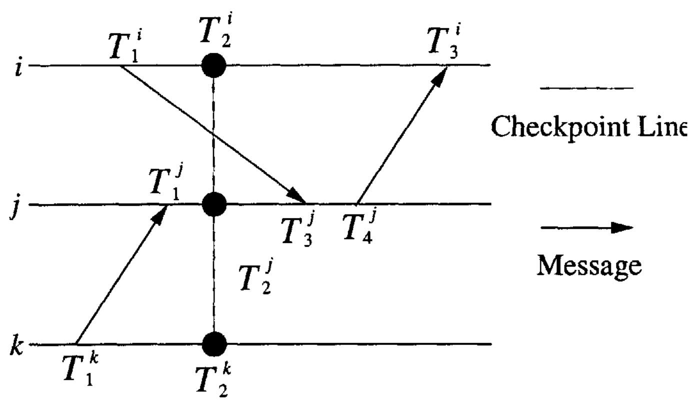


Figure 1-6: Global State saved by Chandy Lamport Algorithm.

after j saved its state. Node i does not save this message because i received it after receiving the marker from j. i's state is correct because in the global state formed by the checkpoints, $Msg_{j \to i}$ has not been sent yet.

The checkpoints created by the Chandy Lamport algorithm form a global state which could have existed in the system because events at different nodes are only partially ordered. The snapshot marker sent from j to i in Figure 1-5 defines an ordering such that all events at j before T_2^j must occur before all events at i after T_2^i. If the snapshot marker took no time to travel from j to i, then the state saved at T_2^i and T_2^j would be saved at the same instant. Similarly, the same can be said between T_2^j and T_2^k. Because processor clocks may operate at slightly different speeds causing time to be relative, the global state formed by the checkpoints at T_2^i, T_2^j, and T_2^k is a valid instantaneous global state (Figure 1-6). In this diagram it is more obvious that $Msg_{i \to j}$ should be saved with the checkpoint because it was in the network at the time that the nodes' state was saved.

The Chandy Lamport algorithm saves the global state of the system with the assumption that no failures occur during the process. If one of the nodes fails before saving its state to stable storage, then it must roll back to its previous saved checkpoint instead of continuing the snapshot and creating a new checkpoint. If any of the nodes in the snapshot is unable to save its state, then all of the nodes in the snapshot must cancel their checkpoint. Nodes can't finalize their checkpoint until they have heard that all other nodes in the snapshot have saved their state to stable storage.

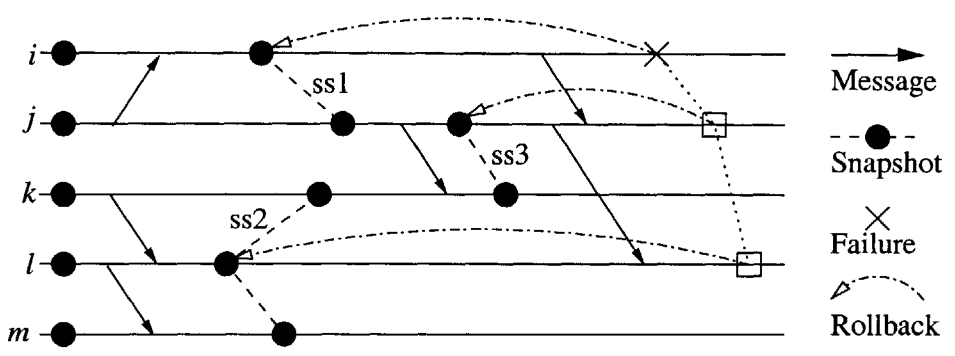


Figure 1-7: Multiple Partial Snapshots and Rollbacks.

Typically, snapshot algorithms use a two phase commit protocol when creating new checkpoints. The first phase tells all nodes to save their state and collects replies from nodes. The second phase tells all nodes to replace the previous checkpoint with the new checkpoint.

1.4 Partial System Snapshots

Instead of saving the state of all nodes in the system at once, nodes can be separated into smaller snapshot groups based on their communication patterns. Figure 1-7 shows independent partial snapshots creating new checkpoints, increasing the amount of work saved in the global state of the system. Nodes i and j create snapshot 1 as a result of j sending $Msg_{j \to i}$. Nodes k, l, and m create snapshot 2 after $Msg_{k \to l}$ and $Msg_{l \to m}$. Later, j sends $Msg_{j \to k}$ and snapshot 3 is created between j and k. After sending $Msg_{i \to j}$, i's application process fails and a rollback begins. Node i restores to its previous checkpoint, which was created in snapshot 1.

When i rolls back, it loses its record of having sent $Msg_{i \to j}$. The system will be inconsistent until j rolls back to a time in which it had not received $Msg_{i \to j}$. j restores to its previous checkpoint created during snapshot 3. However, when j rolls back to this point, it no longer has a record of sending $Msg_{j \to l}$. To complete the rollback, l must restore to its previous checkpoint from snapshot 2, where it had not received $Msg_{j \to l}$. After these nodes finish their rollback, the system will be consistent again. No other nodes will roll back because no other records of sending or receiving a message were lost by the rollbacks of i, j, and l.

Chapter 2

Partial System Snapshots and Rollbacks

Partial snapshots and rollbacks offer a substantial performance improvement over the Chandy Lamport algorithm. The Chandy Lamport algorithm saves a new checkpoint at all nodes and sends a marker between every pair of nodes to save messages in the network. This algorithm is not practical for large systems because the number of markers grows as the square of the number of nodes. Instead of saving every node at once, smaller groups in the system should save their state separately. A group of nodes can checkpoint by themselves if they have not communicated with any other nodes outside of the group since the last checkpoint.

The cost of snapshotting a group of nodes increases as the size of the group increases because every node must hear from all nodes in the system. If this group is the entire system, then increasing the system size will increase the amount of work each node must do for a snapshot. Instead, if the group size is a fixed subset of the entire system, then the system can continue to grow without increasing the snapshot overhead. By similar means, partial system rollbacks improve the system efficiency by reducing the amount of work lost after a failure.

While partial system snapshots and rollbacks are more efficient, they introduce complications that are not present in full system snapshots and rollbacks. The members of the group are not known until the snapshot or rollback is ready to complete.

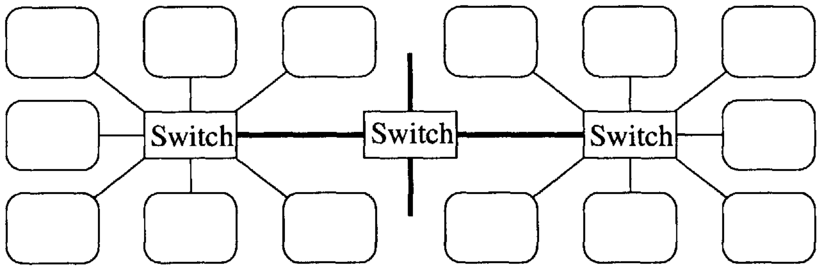


Figure 2-1: Fully connected network using a hierarchy of switches.

After saving its checkpoint, nodes must not send a message to a node that could receive the message before creating a checkpoint in the same snapshot. In addition, two snapshots or rollbacks may begin at separate nodes and then later intersect at common nodes. These intersections need to be handled carefully to ensure a consistent global state without introducing dead locks.

2.1 Node Connectivity

Partial snapshot groups are defined by the communication between nodes. If a node saves its state to a checkpoint, then all other nodes from which it has received messages must also create a checkpoint. A group of nodes can create a checkpoint by themselves if they have not communicated with any other nodes. Nodes can remain independent of each other if the network does not allow messages to be sent between them or if the application's communication style does not send a message between them.

2.1.1 Physical Network Connectivity

The physical network is the first layer of restriction on a node's ability to send messages to other nodes. In a fully connected network, such as a cluster of machines connected by network switches (Figure 2-1), any node can directly send a message to any other node. The only restrictions on message passing will be self imposed by the application process.

Alternatively, each node may have connections to a limited number of other nodes. An extreme example of this case is a string of nodes that can only talk to the two

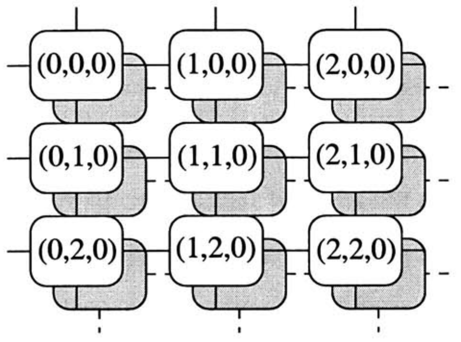


Figure 2-2: Cubic lattice network. Gray nodes are z=1 level.

nodes upstream and downstream from itself. Many systems use a more moderate form of this connectivity in which each node connects to six other nodes. The layout of the network forms a three dimensional cubic lattice. The nodes at the edges wrap around, such that if the nodes had x, y, z coordinates, the nodes at the max positive x positions would connect to the nodes with the same y, z coordinates and the minimum x position (Figure 2-2).

Network layouts can range anywhere between these two cases. Some networks may have clusters of computers that can talk to everyone in the cluster, but only have limited connectivity outside of the cluster. In other networks, nodes may be partitioned to work on different jobs, isolating them from communicating with each other during one specific job and then allowing them to talk freely during a different job [1].

2.1.2 Connectivity due to Application Behavior

Inside the network provided by the physical communication links, the effective connectivity is further limited to the links that have had messages sent over them. If a node i and node j have never communicated directly or indirectly when a snapshot begins, they can be treated as if they were physically disconnected in the period prior to the snapshot. After finishing a snapshot, the dependencies between nodes are reset because all previous dependencies were saved in the snapshot. If a node were to fail and rollback during this period, it would not have to rollback other nodes because it has not communicated with any other nodes.

Nodes' connectivity growth depends on the application's communication style. One possible mode is that nodes communicate often inside of a small group for one phase of computation, then switch and form a group with completely different members in the next phase. If a snapshot is not taken before the a phase switch, then all nodes that are now in a group with a member of the original group are connected. If a system has multiple phases in this style, then the connectivity growth is exponential over these phase changes

An alternative communication model is that every message sent has a probability of being sent to an old node or a new node that the sender has not previously communicated with. This connected set is more difficult to determine in advance because it may be random and can jump abruptly when nodes from two separate sets communicate with each other.

2.2 Cost of Full System Snapshots

The cost incurred in snapshotting is the average time at each node from joining the snapshot and initially saving to finalizing the checkpoint. As described in section 1.3.2, snapshots typically require a phase of contacting every node and collecting replies before checkpoints can be finalized. In parallel computing terms, this can be described as a barrier sync, a save state, and another barrier sync command. The most significant factor contributing to the delay in this stage is likely to be either the network latency between nodes or the delay in saving the node's state to permanent storage. After the initiating node has collected all replies, it sends a commit message to nodes allowing them to finalize their checkpoints.

If nodes share their permanent storage, then the pipe to it may become a bottleneck to the snapshot process. The total delay between joining the snapshot and finalizing the checkpoint will be proportional to the number of nodes that have to use the same storage. The ith node to join a snapshot of N nodes must wait for (N - i) nodes to save their state and send snapshot markers. If the time required for one

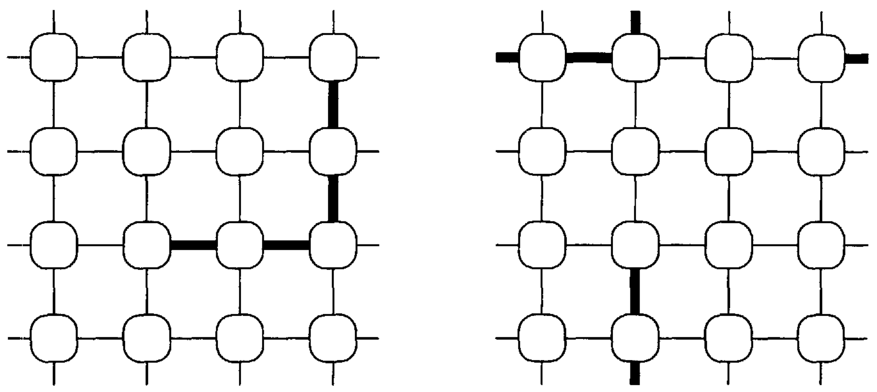


Figure 2-3: The longest path in a 4x4 lattice system.

node to save its state over the pipe to the storage is d, then the average delay per node is on the order of $\frac{1}{2}N * d$.

If each node's permanent storage is either a local device or a shared device with enough throughput that it is not a bottleneck, then nodes will have a uniform, fixed delay before sending a snapshot marker on to other nodes. This delay may also consist of the network latency of sending a marker to another node. The total cost per node from joining the snapshot to finalizing the checkpoint will depend on the longest path between any two nodes i and j in the snapshot. Before a node j can complete its checkpoint, it must receive a notification that i has saved its state. In the lattice described in section 2.1.1, the total longest path is the sum of the longest possible path in each direction (Figure 2-3). If the dimensions are X, Y, Z, then the longest path in each direction is $\frac{1}{2}X$, $\frac{1}{2}Y$, $\frac{1}{2}Z$ because the network wraps around. If the network latency is l and the disk latency is d, then the average cost per node is on the order of $d * l * (\frac{1}{2}X + \frac{1}{2}Y + \frac{1}{2}Z)$. If the dimensions of the lattice are equal, then the cost per node grows on the order of $d * l * \frac{3}{2}X$. Each dimension's size is then calculated as $X = N^{\frac{1}{3}}$. The cost per node for the snapshot thus grows as the cube root of the number of nodes.

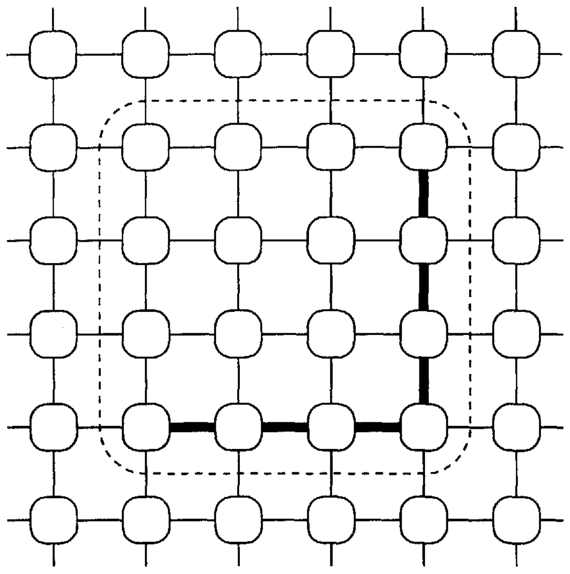


Figure 2-4: The longest path in a 4x4 group in a larger lattice.

2.3 Cost of Partial System Snapshots

The problem with the previous two examples of system snapshot costs and with all full system coordinated checkpoint algorithms is that all nodes must be involved in the process of creating a new set of checkpoints. Each node must receive notification that every other node in the system has saved its state. Even if there is no network or disk latency, the processing demand at each node will increase as the system grows, which will reduce the usable work time in the system.

The cost equations for partial system snapshots are mostly the same, but change slightly because the network doesn't wrap around from one side of the group to the other as it does in the system. This change only causes a constant scale factor difference in the cost of snapshots. A snapshot of a small group in the system is faster and costs less than a snapshot of the entire system because it has fewer nodes to coordinate (Figure 2-4). If the number of nodes in a group is limited to n, then for the lattice system example, the snapshot cost per node is about $d*l*3*n^{\frac{1}{3}}$. The ratio of a group snapshot to a full system snapshot reduces to $(\frac{n}{N})^{\frac{1}{3}}$. If nodes in a lattice system work in three phases where they only communicate along one dimension in each phase, then the group size is X. A snapshot of one of these groups only takes

$\frac{X}{X^3}^{\frac{1}{3}}$ or $X^{-\frac{2}{3}}$. For example, if there are 27 nodes in a group ($X = 27$), then the cost per node for a group snapshot is about $\frac{1}{9}$th the cost per node for a full system snapshot.

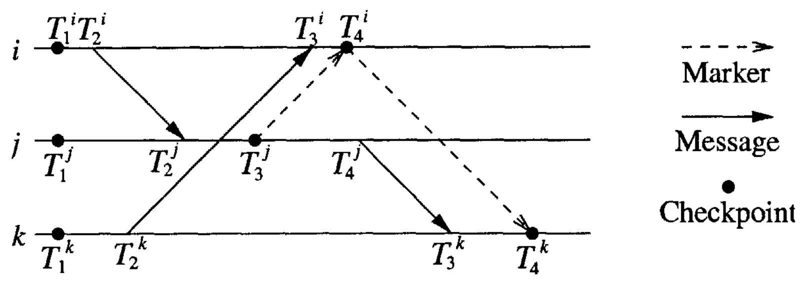
For the system layout that had a shared stable storage at its main bottleneck, the groups can snapshot at different times, reducing the load on the stable storage. The cost per node in a group of n nodes is $\frac{1}{2}n * d$. The ratio of group cost to system cost for this case is $\frac{n}{N}$. For the lattice system with 27 nodes in each group, the cost per node is $\frac{27}{27^3} = \frac{1}{729}$th the cost per node for a system snapshot.

The reduced cost of using partial system snapshots instead of full system snapshots can be used to make the system produce more useful work per node or to reduce the cost of the system. If the same snapshot frequency is used, then the snapshot overhead is reduced and the system will produce more useful work. Alternatively, the snapshot frequency of groups could be increased for the same overhead that was required for the less frequent full system snapshots. More frequent snapshots reduce the amount of work that is lost due to a failure. If the connected group size grows over time as nodes communicate, then the more frequent snapshots will be taken while the groups are smaller. In a system with shared stable storage, the total data sent over the connection to stable storage will remain the same, but the peak usage will be lower if group snapshots are staggered. A smaller, cheaper connection to stable storage can be used instead, or if the same connection is used, the application will have better access to the storage for its own uses.

2.4 Work Lost Due to Failures

In a full system rollback, the entire system resets to the last set of consistent checkpoints. For the lattice system above that has 27 nodes in each of the three directions, 19,683 nodes would rollback. If checkpoints are saved every T_d, then the average work lost per node is $\frac{1}{2} * T_d$. The average work lost for the entire system is $19,683 * \frac{1}{2} * T_d$.

Similar to snapshotting only connected nodes in a system, only connected nodes need to be rolled back after a failure. If the state of a node does not depend on the


i
j
k
T_1^i T_2^i
T_3^i T_4^i
T_1^j T_2^j T_3^j T_4^j
T_1^k T_2^k T_3^k T_4^k
Marker
Message
Checkpoint
Figure 2-5: A message sent after the snapshot that is received before the snapshot.

lost state of a rolled back node, then it does not need to rollback. A proof of the correctness of partial system roll backs is given in section 4.1.3.

If the size of the connected group is n when a failure occurs, then at most those n nodes will need to roll back. The ratio of lost work in a partial system rollback to a full system rollback is $\frac{n}{N}$. In the lattice system model, if snapshots are performed several times per communication phase, then a failure will usually only cause 27 nodes to roll back. The work lost after a failure is reduced to $\frac{1}{729}$th what it could have cost if the entire system needed to roll back. Because the lattice model changes the node grouping after every communication phase, if a failure occurs after a phase change and before a new snapshot, then the connected group size would be 729 and the work lost would be $\frac{1}{27}$th of the work lost by a full system rollback. If snapshots are taken several times or more per phase, then the chance of a failure during this time is reduced.

System models besides the lattice may have varying connected group sizes depending on how recently a snapshot was performed. If the snapshots in the system are partial system snapshots, then the connected group size is limited to the size of the snapshot groups. As stated before, partial system snapshots can be taken more often, reducing the connected group size and the amount of lost work per node.

2.5 Sending Messages During Snapshot

Partial system snapshot algorithms have a complication that nodes in a snapshot don't know whether some other nodes are in the same snapshot or not. If a node sends a message after saving the application state to a tentative checkpoint, the message must not be saved before another node in the same snapshot creates a checkpoint. In Figure 2-5, Node j initiates a snapshot at T_3^j, and sends a marker to i. At T_4^j, j sends an application message to k. Meanwhile, i receives the marker from j, saves its local state, and sends a marker to k because it received a message from k at T_3^i. In this situation, it is possible that k will receive the message from j before receiving the marker from i. When k saves its state, it will have already received the application message from j. The global state formed by these checkpoints would show that k has received an application message from j that j has not sent.

In the Chandy Lamport algorithm, all nodes are a member of the same snapshot and are sent a marker that separates messages before a checkpoint from messages after a checkpoint. One approach to this problem for partial system snapshots is to not allow nodes to send messages from when they create a tentative checkpoint until all of the members of the group are known. This mechanism was used in Koo and Toueg's algorithm [6], which will be discussed further in Section 3.1. Blocking the application from sending messages during a snapshot decreases the efficiency of the system. Instead of interfering with the application, Moriya and Araragi's algorithm [7] (discussed in section 3.2) sends a marker before before messages are sent to new nodes during a snapshot.

2.6 Intersections between Groups

An intersection between groups occurs when one node that is already in a snapshot or rollback group receives a marker from a different snapshot or rollback. To show this in terms of a system execution, let an event at any node be represented by e_i. A system execution can be given by ordering all of these events into a single sequence $e_0, e_1, e_2, e_3 \ldots e_n$.

In Figure 2-6, snapshot 1 includes nodes i, j, and k. Snapshot 2 includes nodes l, m, and n. Two snapshots that have time overlap are running on different nodes resulting in this not being an intersection. The two snapshots run to completion

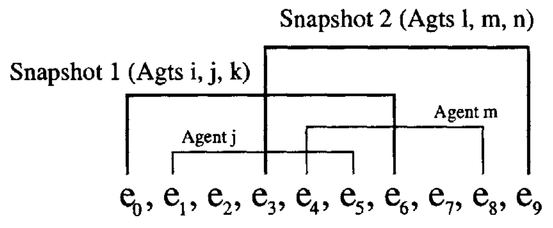

Snapshot 2 (Agts l, m, n)
Snapshot 1 (Agts i, j, k)
Agent j
Agent m
e0, e1, e2, e3, e4, e5, e6, e7, e8, e9

Figure 2-6: Two snapshots occurring at the same time without intersecting.

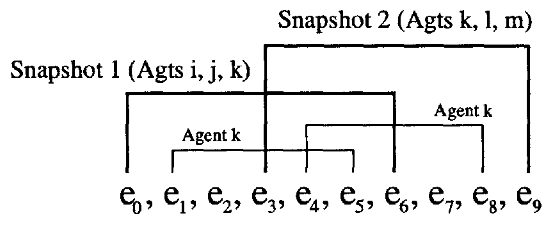

Snapshot 2 (Agts k, l, m)
Snapshot 1 (Agts i, j, k)
Agent k
Agent k
e0, e1, e2, e3, e4, e5, e6, e7, e8, e9

Figure 2-7: Two snapshots intersecting at node k.

without using any of the same nodes, thus each one can be treated as an isolated snapshot.

In Figure 2-7, snapshot 1 uses nodes i, j, and k. However, snapshot 2 now includes nodes j, k, and l, creating an overlap at node k. The two snapshots run at k at the same time, creating an intersection. Each snapshot is transitively dependent on the nodes in the other snapshot because each depends on k and its dependencies. Similar intersections can also occur between two rollbacks or between a rollback and a snapshot.

2.6.1 Snapshot-Snapshot Intersections

When two different snapshot groups intersect at an node, the most efficient way to handle their snapshots is to have them create a single set of checkpoints together. If snapshot 1 depends on a node that is in snapshot 2, the initiator for snapshot 1 can depend on the initiator of snapshot 2 to handle that node. As the group of dependent nodes becomes larger, it becomes more likely that multiple nodes will initiate snapshots at the same time.

Where a snapshot refers to a group of nodes with one initiator, a super-snapshot is a set of snapshot groups with one coordinator. A coordinator is an initiator that

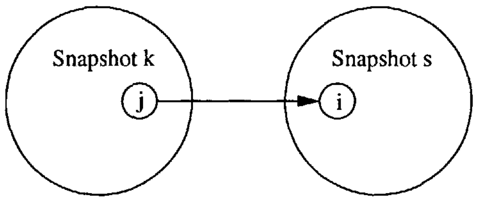


Figure 2-8: $SS^k_{j \to i}$ makes Snapshot k Dependent on Snapshot s.

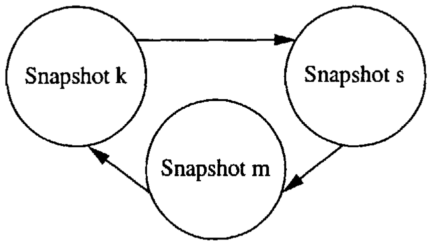


Figure 2-9: Cyclic Snapshot Dependencies.

has been chosen to collect the status of the intersected snapshots and manage their dependencies. It takes the role of a higher level central control point for the super-snapshot. It tells snapshots to complete when all of the other snapshots that they depend on have reached tentative complete state.

Tentative complete state in a snapshot group is reached when the members of the group have been completely determined and all of them have saved their state to stable storage. Before this condition, it is still possible that one of the nodes in the snapshot group may be unable to save its state and complete the snapshot. Because the snapshot may not be able to complete, no node in this snapshot or in a snapshot that depends on this snapshot can complete. If all other snapshots that this snapshot depends on have reached tentative complete state, then this snapshot is guaranteed to complete.

When node i in snapshot s receives $SS^k_{j \to i}$, snapshot k becomes dependent on snapshot s to complete, as shown in Figure 2-8. This dependency is only in one direction. If snapshot s is unable to complete, then snapshot k will be unable to complete. However, snapshot s can complete whether or not snapshot k completes. If another node in snapshot k receives a snapshot marker from a node in snapshot s, then a dependency would exist in both directions. In that case, if either of the snapshots was unable to complete then neither can complete.

A more complicated set of dependencies is shown in Figure 2-9. Snapshots s, k, and m are each dependent on one other snapshot, creating a cycle of dependencies. For snapshot k to complete, snapshot s and all snapshots that s directly or transitively depends on must reach tentative complete state. The coordinator can detect and properly handle these dependency cycles because it collects and manages every snapshot's dependencies.

2.6.2 Rollback-Rollback Intersections

The intersection handling of two rollback groups is simpler than snapshot-snapshot intersections because rollbacks will always complete. When two rollbacks intersect, the most efficient action is to delegate the dependencies of a node that is common to multiple rollbacks to the rollback from which it received a marker first. Nodes that have joined multiple groups resume their application process after all rollbacks that they have joined have completed.

2.6.3 Snapshot-Rollback Intersections

The intersection of a snapshot and a rollback implies that one of the nodes necessary to the snapshot has failed and is unable to save its application state. In this case, the snapshot must be canceled to allow the rollback to proceed. Canceling a snapshot discards the tentative checkpoint and returns the node to the normal operating state. Nodes that receive a rollback marker will join the rollback after they finish canceling the snapshot. If the canceled snapshot was part of a super-snapshot, its initiator informs the coordinator of the cancellation. The coordinator then cancels all other snapshots that depended on the canceled snapshot. If a snapshot in the super-snapshot did not depend on the canceled snapshot, then it will continue its snapshot. After a snapshot-rollback intersection has been handled, only the rollback will continue. In the case of a rollback intersection with a super-snapshot, an isolated part of the super-snapshot may also remain if it is not interacting with the rollback group.

This procedure is the simplest way to handle snapshot-rollback interactions, but it is not the most efficient. In some cases, it is not necessary to cancel the snapshot. If a node i joins a snapshot and then receives $RB^k_{j \to i}$ from an node j that it doesn't depend on for the checkpoint, then i can still successfully complete its local checkpoint and the snapshot group of which it is a member. Node i can handle $RB^k_{j \to i}$ and join the rollback after the snapshot finishes.

Chapter 3

Partial System Snapshot Implementations

The Koo Toueg algorithm and the Moriya Araragi algorithm create partial system snapshots, but neither handles intersections between snapshot and rollback groups efficiently. The Koo Toueg algorithm blocks the application from sending messages during a snapshot to guarantee consistent checkpoints and cancels groups that intersect. The algorithm created by Moriya and Araragi allows the application to send markers during snapshot by sending a marker before messages to nodes that haven't already been sent a marker. This change allows their algorithm to be more efficient, but the algorithm didn't explain how it handled intersections between groups. Their algorithm is extended to allow intersecting snapshot and rollback groups to merge and continue when possible.

3.1 Koo and Toueg's Algorithm

Koo and Toueg's algorithm [6] for partial system snapshots is well suited to a lattice type system. Each node keeps one finalized checkpoint and stores one tentative checkpoint while it is running the snapshot process. The nodes remember which other nodes they have communicated with since the previous checkpoint. One node begins the snapshot, saves its local state in a tentative checkpoint, sends markers to all of the nodes it has received message from since the last checkpoint, and waits for replies from all of those nodes. Each node that receives a marker follows the same process of saving state, sending markers, and waiting for replies. When a node has received positive replies from all of the nodes that it sent markers, then it replies positively. For all markers received after the first marker, the node replies positively because it has no new markers that it needs to send. When the node that began the snapshot receives positive replies from all of the nodes that it sent markers, it makes the tentative checkpoint permanent, replacing the previous checkpoint. The initiator then sends a commit message that spreads through the group in the same manner as the markers did.

If one of the nodes fails during the snapshot, then it either returns a negative reply to the snapshot marker, or the node that sent it a marker detects its failure and replies negatively to the marker it received. The negative reply continues up to the node that initiated the snapshot. That node sends an undo message to all of the nodes, which makes them erase their tentative checkpoint. From the time that nodes save their state, until the time that they make the checkpoint permanent or undo the checkpoint, they are not allowed to send any application messages. This restriction keeps nodes from sending a message after they checkpoint which could be received by another node before it checkpoints. This issue will be further explained in the next section.

3.2 Moriya and Araragi's Algorithm

Their implementation is intended to run on a fully connected network. The message transportation is first in first out and reliable. Messages sent from one node to another always arrive after a finite delay. Each node consists of a monitor process and an application process. The monitor process controls the fault tolerance algorithms. The monitor process never fails, even if the application process has failed. The monitor process maintains a list of nodes that it has sent a message to or received a message from. In addition to saving the node's state during a snapshot, the monitor process all saves all messages that were received during the snapshot from other nodes in the group. This maintains an appearance of a reliable network to the application process.

3.2.1 Snapshots

A node i may begin a new snapshot and become its initiator due to a timer expiring or reaching a threshold, such as some number of nodes in the dependency list. Nodes join a snapshot when they receive a snapshot marker and have not already joined the snapshot. The monitor process saves the state of the application process and then sends snapshot markers, $SS^i_{i\to j}$, to all nodes j in its dependency list, $D^i_{[P,C]}$. While the monitor process is performing a snapshot, the application process continues sending, receiving, and processing application messages. During the snapshot, if a node i sends an application message $Msg_{i\to k}$ to node k and has not already sent $SS^i_{i\to k}$, then the marker is sent before the message. This marker ensures that the application message that was sent after $T^i_{C_S}$ is not received before k joins that snapshot at $T^k_{C_S}$.

After saving their local state and sending snapshot markers, nodes send a my_group_set message to the initiator. This message contains $D^j_{[P,C]}$. When initiator i receives a my_group_set message from node j, it adds j to the list of nodes that have joined the snapshot, G^i_{HJ}, and adds $D^j_{[P,C]}$ to the list of nodes that must join the snapshot for the snapshot to complete, G^i_{MJ}.

Moriya and Araragi addressed sending messages during a snapshot (Section 2.5) by always sending a snapshot marker before the message if a marker has not already been sent to the receiving node. The application process operates normally during a snapshot and is allowed to send messages to other nodes. This approach decreases the overhead in comparison to Koo and Toueg's system where the application is not allowed to send any messages during a snapshot. However, these markers cause additional complication because the recipient is not always a member of the group. These markers may even be sent after the initiator has completed the snapshot. To handle them, the initiator can tell those nodes that are not in the snapshot to cancel the checkpoint because none of the nodes in the group were dependent on them.

The snapshot continues until $G^i_{HJ} \supseteq G^i_{MJ}$. When this condition is met, all nodes that were depended on directly or indirectly by a member of the snapshot have joined the snapshot. All of the nodes have saved their application state and sent all snapshot markers. The initiator, i, then sends marker_set messages to all nodes $j \in G^i_{HJ}$. This message contains the list of all nodes that have sent j a marker. When j receives the marker_set message and markers from all nodes in that list, it completes its local snapshot. In addition to the application state, all messages from a node i received after $T^j_{C_S}$ and followed by a snapshot marker from i are stored in the checkpoint.

3.2.2 Rollbacks

The rollback procedure is very similar to the snapshot procedure. A monitor process for a node i stops the application process and begins a rollback when it detects a failure or receives a rollback marker. The node that failed and began the rollback becomes the initiator for the rollback group. Nodes send rollback markers, $RB^i_{i \to j}$, to all nodes j in their dependency list, $D^i_{[P,C]}$. Next, the nodes send a my_group_set message, which includes $D^j_{[P,C]}$, to initiator i. When i receives this my_group_set message, it adds j to G^i_{HJ}, and adds $D^j_{[P,C]}$ to G^i_{MJ}.

The rollback completes when $G^i_{HJ} \supseteq G^i_{MJ}$. At this time, all nodes that were directly or indirectly dependent on i have joined the rollback group and sent their markers. Initiator i sends marker_set messages to all nodes $j \in G^i_{MJ}$. The marker_set message contains a list of all nodes that have sent j a rollback marker. When a node j receives a $RB^i_{k \to j}$, it deletes all $Msg_{k \to j}$ that it has received since the beginning of the rollback. Once a node has received markers from all of the nodes listed in the marker_set message, it resumes the application process using the state from the last checkpoint.

3.3 Pseudo Code for the Extension of Moriya and Araragi's Algorithm

The pseudo code is based on the algorithm developed by Moriya and Araragi, with some modifications and some additions to handle the intersections between snapshot and rollback groups. These intersections are handled as described in section 2.6. Intersecting snapshots form a super-snapshot with a coordinator that organizes dependencies between different groups. The algorithm for selecting a coordinator from intersecting snapshots and super-snapshots is described in section 3.3.3. Intersections between snapshots and rollbacks cancel the snapshots that depended on failed nodes. Intersecting rollbacks work together at the nodes where they intersect.

3.3.1 General Procedures

Node i receives $Msg_{j \to i}$ from node j

```
1. If running snapshot or rollback
   (a) Save $Msg_{i \to j}$ in queue($j$)
```

Node i sends $Msg_{i \to j}$ to node j

```
1. If running snapshot $k$ and node $j \notin D^i_{[P,-]}$
   (a) Send $SS^k_{i \to j}$
```

During a rollback, received messages are queued to be handled after the application process resumes if the messages are still valid. In a snapshot, messages are saved in a queue to later be saved with the checkpoint if they are followed by a snapshot marker. In both of these cases, the validity of a message, or whether it needs to be saved can not be determined until a marker from the same sender has arrived or the snapshot or rollback has completed. During a snapshot, the message is passed on to the application process immediately.

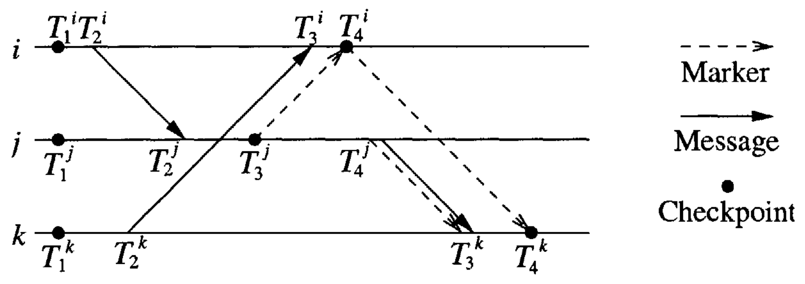


Figure 3-1: Sending a marker before a message.

If j is performing a snapshot and sends a message to a node k that j has not communicated with since the previous snapshot, then a marker is sent before this message. If k is not in $D^j_{[P,-]}$ when the message is about to be sent, then j has not sent $SS^j_{j\to k}$. In figure 3-1, $SS^j_{j\to k}$ sent at T^j_4 guarantees that k will join the snapshot before receiving $Msg_{j\to k}$.

3.3.2 Snapshot Procedures

Node k starts snapshot

1. k becomes initiator of snapshot
2. Save application process state to temporary checkpoint
3. Send $SS^k_{k\to j}$ to $\forall j \in D^k_{[P,C]}$

When a monitor process begins a snapshot, it becomes the initiator of the snapshot. The monitor process saves the application process state to a temporary checkpoint and sends snapshot markers to all nodes that have communicated with it since the last checkpoint.

Node i receives marker $SS^k_{j\to i}$ from node j for the snapshot started by k

1. Add j to received_markers
2. If i is not performing a snapshot or rollback
   (a) Save application process state to temporary checkpoint
   (b) Send $SS^k_{i\to l}$ to $\forall l \in D^i_{[P,C]}$

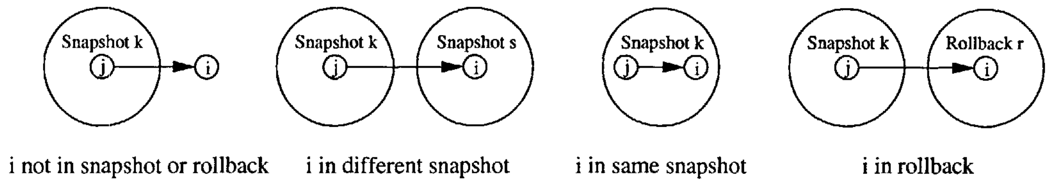


Figure 3-2: The four cases of receiving a snapshot marker.

```
    (c) Send my_group_set($i$, $D^i_{[P,C]}$, join) to $k$
    (d) Add $k$ to joined_snapshots
3. Elsif $i$ is performing a snapshot
    (a) If $i$ has already joined the snapshot started by $k$
        i. Check_Snapshot_Finished()
    (b) Elsif $i$ has not joined the snapshot started by $k$
        i. Send initiator_dependency( joined_snapshots) to $k$
        ii. Send my_group_set($i$, $\emptyset$, join) to $k$
        iii. Add $k$ to joined_snapshots
4. Elsif $i$ is performing a rollback
    (a) Send my_group_set($i$, $\emptyset$, cancel) to $k$
```

Check_Snapshot_Finished()

```
1. If received_marker_sets $\supseteq$ joined_snapshots and received_markers $\supseteq$ expected_markers
    (a) If cancel_snapshot is not true
        i. Save all $Msg_{j \to i}$ that are followed by $SS^k_{j \to i}$ for any $k$ in joined_snapshots
           with temporary checkpoint
        ii. Replace permanent checkpoint with temporary checkpoint
    (b) If cancel_snapshot is true
        i. Discard all snapshot state, return to normal state
    (c) Terminate snapshot procedure
```

The four different cases for handling a snapshot marker are shown in Figure 3-2. When a node i receives $SS^k_{j\to i}$, if it is not a member of a snapshot or rollback group, it joins the snapshot. The monitor process saves the application process state and sends $SS^k_{i\to l}$ to all of the nodes it has communicated with since the beginning of the last snapshot. Node i sends a my_group_set message to initiator k, which says that i has joined the snapshot and saved its state. The message also gives k the list of nodes that i depends on and has sent markers to.

The following procedures are the extensions to the algorithm that handle the cases when the node is already a member of a snapshot or rollback. If i was already in this marker's snapshot group, it checks to see if it can finish the snapshot after receiving this marker. Before it can complete, it must receive a marker_set from the initiator of every snapshot it has joined and it must receive markers from all nodes listed in these sets. When these conditions are met, the snapshot either completes or cancels according to the cancel flag in the marker_set message that it received earlier. To complete the snapshot, i saves all messages that arrived after the start of the snapshot and were followed by a snapshot marker from the message sender. The temporary checkpoint then replaces the previous permanent checkpoint.

If i was a participant in a different snapshot than this marker, i sends an initiator_dependency message to the new initiator, k, with a list of the snapshots at i that snapshot k depends on. Then i joins this snapshot by sending a my_group_set message to initiator k. The dependency list in this message does not contain any new nodes because markers have already been sent to all nodes that i depends on.

If i was performing a rollback, then the snapshot started by k must be canceled because i can not save its application process state. Node i sends a "cancel" my_group_set message to initiator k. The dependency list in the message is empty because i does not send out new markers because the snapshot will be canceled.

Initiator k receives my_group_set(i, $D^i_{[P,C]}$, join_or_cancel)

1. Add i to G^k_{HJ}

2. Add $D^i_{[P,C]}$ to G^k_{MJ}

```
3. If join_or_cancel is cancel
   (a) Set cancel_snapshot to true
4. Initiator_Check_Snapshot_Finished()
```

Initiator_Check_Snapshot_Finished()

```
1. If $G^k_{HJ} \supseteq G^k_{MJ}$
   (a) If cancel_snapshot is not true
       i. If this snapshot is part of a super-snapshot
          A. Send tentative_complete_reached($k$) to coordinator
       ii. Elsif this snapshot is not part of a super-snapshot
          A. Send marker_set($k$, marker_list, complete) to $\forall j \in G^k_{HJ}$
   (b) Elsif cancel_snapshot is true
       i. Send marker_set($k$, marker_list, cancel) to $\forall j \in G^k_{HJ}$
       ii. Send snapshot_canceled($k$) to coordinator
```

When initiator k receives a my_group_set() message, it adds i to the list of nodes that have joined, G^k_{HJ}. The nodes in the dependency list are added to G^k_{MJ}, the list of nodes that must join the snapshot before it can finish. If the message has the cancel flag set, then the snapshot is marked to cancel when all nodes have joined. The initiator then checks if the snapshot is ready to be finished.

When all required nodes have joined the snapshot ($G^k_{HJ} \supseteq G^k_{MJ}$), it has either reached tentative complete state or is ready to cancel. If no nodes have canceled and k is part of a super-snapshot, then k tells the coordinator that it has reached tentative complete state. Initiator k waits for the coordinator to send a message saying that all of the snapshots k depends on have completed. If k is not part of a super-snapshot and does not cancel, then it sends "complete" marker_sets to every node in G^k_{HJ}. The marker_set contains a list of every node that has sent a marker to j and tells j to complete its snapshot when all of those markers are received.

If k has received a cancel message, it sends "cancel" marker_sets to all nodes in G_{HJ}^{k}. These marker sets contain a list of all nodes that have sent j a snapshot marker. The marker set tells j to cancel its snapshot once it receives all of the markers that have been sent to it. If k was part of a super-snapshot, then a snapshot_canceled message is sent to the coordinator.

Node i receives marker_set(k, marker_list, cancel_or_complete)

1. Add k to received_marker_sets
2. Add marker_list to expected_markers
3. If cancel_or_complete is cancel
   (a) Set cancel_snapshot to true
4. Check_Snapshot_Finished()

The procedure for handling a marker set is the same as the original algorithm in Moriya and Araragi's system, except that the marker set also carries a flag for cancel or complete in addition to the list of nodes that have sent markers to i. After receiving a marker_set, i checks if it can now finish the snapshot.

Initiator k receives initiator_dependency(intersected_snapshots)

1. Add intersected_snapshots to snapshot_dependency_list
2. Update coordinator selection algorithm
3. Send new_dependencies(k, intersected_snapshots) to coordinator

Coordinator receives new_dependencies(k, snapshots_dependent_on)

1. Add depends(k, snapshots_dependent_on) to super_snapshot_dependencies

These rest of the procedures in the snapshot section have been added to handle communication with the coordinator and dependencies between snapshots. When initiator k receives an initiator_dependency message, it adds the list of intersected snapshots to the list of snapshots that it directly depends on to complete. If neither

k nor the intersected snapshots had a coordinator, then one is chosen from those initiators. If one or more already existed, then the most suitable one is chosen according to the coordinator selection algorithm. One algorithm for choosing a coordinator is provided in section 3.3.3. After the coordinator has been determined, k's list of dependencies is sent to it. When the coordinator receives the new_dependencies message, it adds k's list of dependencies to super_snapshot_dependencies.

Coordinator receives snapshot_canceled(k)

1. Send cancel_request to all snapshots dependent on k

Initiator k receives cancel_request

1. Set cancel_snapshot to true
2. Initiator_Check_Snapshot_Finished()

When the coordinator receives a snapshot_canceled message, it sends a cancel_request message to all snapshots that directly or transitively depend on the canceled snapshot k. When initiators receive this message, they mark their snapshot to cancel and then check if the snapshot is ready to finish.

Coordinator receives tentative_complete_reached(k)

1. If any of k's dependencies canceled
   (a) Send cancel_request to k
2. Send dependencies_met to initiators that have had all of their dependencies reach tentative complete state

Initiator k receives dependencies_met(completed_snapshots)

1. If completed_snapshots $\supseteq$ snapshot_dependency_list
   (a) Send marker_set(k, marker_list, complete) to $\forall j \in G_{HJ}^{k}$

When the coordinator receives a tentative_complete_reached message, it checks through super_snapshot_dependencies for any snapshots that are tentative complete and have had all of their direct and transitive dependencies met. All snapshots that have had their dependencies met are sent a dependencies_met message. This message contains the list of all snapshots in the super-snapshot that have completed. When initiator k receives the dependencies_met message, if all its dependencies are tentative complete, then it completes the snapshot.

3.3.3 Coordinator Selection Algorithm

The selection of a coordinator should be based on an absolute factor, such as the lowest initiator ID or most available resources. The procedure of choosing a coordinator requires multiple message exchanges to find the initiator with the lowest ID. Another snapshot intersection may occur or coordinator information may be sent while a coordinator is still being selected. All initiators assume that their coordinator is the lowest ID initiator that they are aware of. If an initiator is no longer the coordinator and receives information for the coordinator, that initiator sends the information to the initiator that it believes is the coordinator. When an initiator with a lower ID becomes the coordinator, the old coordinator sends all information that it had accumulated to the new coordinator. This mechanism ensures that information is always sent to an initiator with a lower ID until it reaches the current coordinator.

Coordinator selection begins when an initiator receives an initiator_dependency(intersected_snapshots) message from a node. The initiator sends the intersected snapshots list to its coordinator, or temporarily becomes its own coordinator if it doesn't currently have one. This coordinator will be called k. The initiator in intersected_snapshots with the lowest ID will be called j. If j's ID is lower than k, then k joins j's super-snapshot by sending a join message. That message is forwarded until it reaches the coordinator (which may no longer be j). The coordinator replies back to k with the current coordinator's ID if it is not j. k stops being coordinator and tells all snapshots underneath it to change to this new coordinator.

If k had a lower ID than j, k sends a message to j telling it to change the coordinator to k. Again, if j is no longer the coordinator for that super-snapshot, the message is forwarded until it reaches the coordinator. If the coordinator has a lower ID than k, the coordinator will reply back to k and k will change to this new coordinator. If the coordinator had a higher ID, then it will stop being a coordinator and switch itself and all initiators underneath it to use k as coordinator.

3.3.4 Rollback Procedures

Node k fails and starts rollback

1. k becomes initiator of rollback
2. Stop application process
3. Send $RB^k_{k \to j}$ to $\forall j \in D^k_{[P,C]}$

When the application process at a node fails, the monitor process begins the rollback procedure and becomes the initiator of the rollback. The application process is stopped to prevent it from sending any further invalid messages. Rollback markers are sent to all nodes that k has communicated with since the beginning of the last checkpoint.

Node i receives marker $RB^k_{j \to i}$ from node j for the rollback started by k

1. Add j to received_markers
2. If i is not performing a snapshot or rollback
   (a) Stop application process
   (b) Send $RB^k_{i \to l}$ to $\forall l \in D^i_{[P,C]}$
   (c) Send my_group_set(i, $D^i_{[P,C]}$) to k
   (d) Add k to joined_rollbacks
3. Elsif i is performing a rollback
   (a) If i has already joined the rollback started by k
      i. Check_Rollback_Finished()
   (b) Elsif i has not joined the rollback started by k

```
         i. Send my_group_set($i$, $\emptyset$) to $k$
        ii. Add $k$ to joined_rollbacks
  4. Elsif $i$ is performing a snapshot
    (a) Send cancel message to current snapshot initiator
    (b) Handle rollback marker after snapshot has finished
```

Check_Rollback_Finished()

```
  1. If received_marker_sets $\supseteq$ joined_rollbacks and received_markers $\supseteq$ expected_markers
    (a) Clear all $Msg_{j \to i}$ that are followed by $RB^k_{j \to i}$
    (b) Restore previous permanent checkpoint
    (c) Resume application process
    (d) Terminate rollback procedure
```

In addition to the original single rollback algorithm, the extensions to handle the receipt of a rollback marker while the node is running a snapshot or a different rollback have been added. If node i is not a member of a snapshot or rollback group and receives $RB^k_{j \to i}$, it joins that rollback. The monitor process stops the application process and sends $RB^k_{i \to l}$ to every node it has communicated with since the beginning of its last checkpoint. To complete the join procedure, i sends initiator k a my_group_set message containing the list of nodes i is dependent on.

If the marker is from a rollback that i is already a member of, i checks to see if that was the last marker it needed to complete the rollback. If i has received marker_sets from the initiators of all of the rollbacks it has joined and has received all expected rollback markers, then it completes its rollback. All messages that were followed by a rollback marker from the same sender will be cleared from the queues. The application state will be restored to the previous permanent checkpoint. The application process will be resumed and will start operating on the remaining valid messages in the queues.

If the marker is from a rollback that i is not a member of, i joins that rollback by sending a my_group_set message to k. This message contains an empty list of dependencies because i has already sent rollback markers to all agents that it depended on.

If i is part of a snapshot group, then that snapshot will be canceled because i may have depended on j for the snapshot. Node i will send a cancel message to its initiator. i will handle $RB^k_{j \to i}$ and join the rollback after the snapshot has finished.
Initiator k receives my_group_set(i, $D^i_{[P,C]}$)

1. Add i to G^k_{HJ}

2. Add $D^i_{[P,C]}$ to G^k_{MJ}

3. If $G^k_{HJ} \supseteq G^k_{MJ}$

   (a) Send marker_set to $\forall j \in G^k_{HJ}$

When initiator k receives a my_group_set message, it adds the sender to the list of joined nodes, G^k_{HJ}. The dependency list from the message is added to the list of nodes that must join the rollback, G^k_{MJ}. If all nodes in G^k_{MJ} have joined the rollback, then it completes. Initiator k sends marker_set messages to every node in G^k_{HJ} with a list of markers that they must receive before they can complete.
Node i receives marker_set(k, marker_list)

1. Add k to received_marker_sets

2. Add marker_list to expected_markers

3. Check_Rollback_Finished()

When i receives a marker_set message, it adds k to the list of initiators that it has received a marker_set from. The marker_list is added to the list of expected markers. Node i checks to see if all conditions have been met to complete the rollback.

Chapter 4

Proof of Partial Snapshots and Rollbacks

For the fault tolerance algorithm to be useful, it must lead to a consistent execution of the system and guarantee that work will be completed. The implementation explained in section 3.3 creates checkpoints that form a valid global state of the system. After a failure, these checkpoints are restored and all invalid state in the system from the time of the previous checkpoint until the rollback is removed, returning the system to the global state saved in the checkpoints. These algorithms are deadlock free and will finish in finite time. If the snapshot frequency is chosen to be more often than the failure frequency discussed in section 1.2, then statistically, new snapshots will be created.

4.1 Consistency

Consistency for a system is well explained in the paper by Chandy Lamport [2]. In short, a system state is consistent if all received messages have been sent by some node in the past. The checkpoints created in the snapshot process should form a consistent state of the system or should be discarded if this is not possible. The rollback process should restore these checkpoints and remove all invalid state and messages from the time of the previous checkpoint until the failure.

4.1.1 Partial Snapshots

The proof of correctness of creating snapshots is broken into the following parts. First, during failure free operation, the snapshot process does not alter the application process besides adding finite delays to messages. Second, the group of nodes performing a snapshot create a consistent global state among themselves according to the Chandy Lamport algorithm. Third, after the snapshot is finished, the checkpoints of all of the nodes in the system form a complete global state.

During a snapshot, the monitor process reads the state of the application process, but does not alter it in any way. The monitor sends and receives markers and marker lists. These snapshot messages are pushed onto the network links between application messages. The order of the application messages is not changed. Adding the snapshot messages to the network links will delay the arrival the application messages. These delays may cause application messages from different sources to arrive in a different order. These changes are acceptable because they are the same as delays caused by network congestion.

The group of nodes participating in a snapshot or super-snapshot creates a global state for that group. When a node receives a marker, it joins the snapshot group, saves its local state, and sends markers to all nodes in its dependency list. The dependency list contains all nodes that have been sent a message or that the node received a message from. The snapshot will not complete until all of the nodes in the dependency list of any of the nodes in the group have joined. The group will be correctly decided by the completion of the snapshot because all nodes that have communicated with a group member since the previous checkpoint will have joined the group.

Figure 4-1 shows a marker sent due to an application message received by a group member. Node j saves its state in a previous snapshot at time T_1^j. Node k's previous checkpoint is saved at T_1^k. At T_2^k, k sends $Msg_{k \to j}$ and adds j to $D_{[1,2]}^k$. j receives $Msg_{k \to j}$ at T_2^j and adds k to $D_{[1,2]}^j$. When j joins snapshot i at T_4^j, it sends $SS_{j \to k}^i$ to k because k is in $D_{[1,4]}^j$. This marker forces k to create a new checkpoint at T_3^k. If k

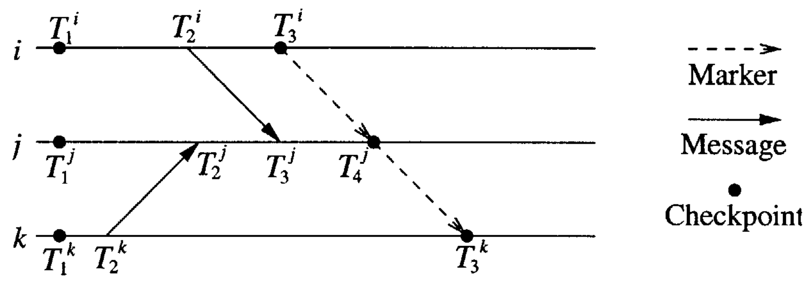


Figure 4-1: Node j sends $SS^i_{j\to k}$ because of $Msg_{k\to j}$.

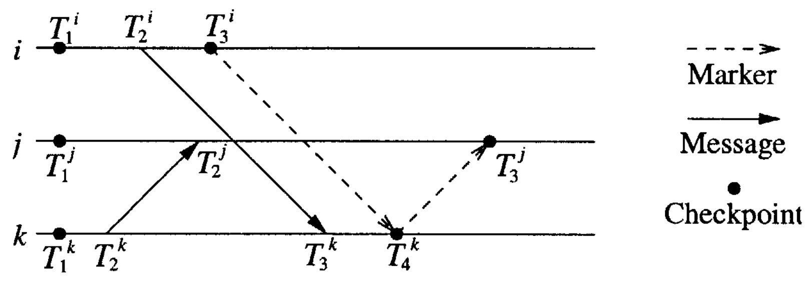


Figure 4-2: Node k sends $SS^i_{k\to j}$ because of $Msg_{k\to j}$.

did not join the snapshot and create a new checkpoint, then a failure would cause k to roll back to T_1^k and j would roll back to T_4^j. This global state would be incorrect because j has received $Msg_{k\to j}$, but k has not sent it.

Alternatively in Figure 4-2, if k joins snapshot i first, k would send $SS^i_{k\to j}$ at T_4^j. j would receive the marker at T_3^j, forcing it to join the snapshot and create a new checkpoint. If j did not join the snapshot, a failure would cause j to roll back to T_1^j and k would roll back to T_4^k. This global state would be incorrect because k has sent $Msg_{k\to j}$, but j will never receive it because the message was not saved. Therefore, if k joins a snapshot, it sends a marker to all nodes that it has received a message from or sent a message to.

Inside of this set of nodes, the snapshot procedure produces the same global state as the CL model would produce among these nodes. Every node saves its state after the first snapshot marker it receives. As shown in the example above, each node j that has sent $Msg_{j\to i}$ added i to its dependency list. When j joins the snapshot, it sends $SS^k_{j\to i}$ to all nodes i in $D^j_{[P,C]}$. If i joined the snapshot before receiving $SS^k_{j\to i}$, then it saves $Msg_{j\to i}$ with the checkpoint. This procedure sends the same markers that the CL model would send except that markers are only sent over network links that have had application messages sent over them. If a message has not been sent over a network link, then a marker is not necessary because there are no messages on that link that need to be saved.

Each node sends the initiator the set of the markers that it sent. The initiator collects these sets and sends a list to each node containing the markers that it will receive. The initiator does not send these lists until it has received a marker set from every node that has been sent a marker. Each node must receive at least one marker and successfully save its local state before any of the nodes can complete the snapshot (Section 1.3.2). When a node receives all of the markers on the list, it will have saved all of the messages that were in its inbound message links at the time when the application state was saved. When all nodes have completed the snapshot, the state of all of the nodes and all of the network links between them will be saved, creating a complete global state for that group.

The previous proofs also apply to the case of super-snapshots. Instead of a single group with one initiator, a super-snapshot has multiple groups, each with its own initiator, and a single coordinator. The coordinator delays the completion of any of the groups until all of the dependent groups have received marker sets from all of the node that were sent markers by members of that group. If all of the initiators have received all of these marker sets, then every node has received at least one of the markers from each snapshot group that sent it markers. The nodes will wait to receive marker lists and markers from every snapshot they have joined before they complete their snapshot. Therefore, the nodes in the super-snapshot each complete their local snapshot when they have received all markers that were sent to them by any node in the super-snapshot group. The checkpoints created by the nodes will form a complete global state covering all of the nodes in the super-snapshot.

The checkpoints created by a snapshot or super-snapshot form a global state of the entire system when combined with the existing checkpoints of the nodes outside of the group. In a distributed system without synchronized clocks, the time at different nodes is relative, restricted only by the partial orders created by communication

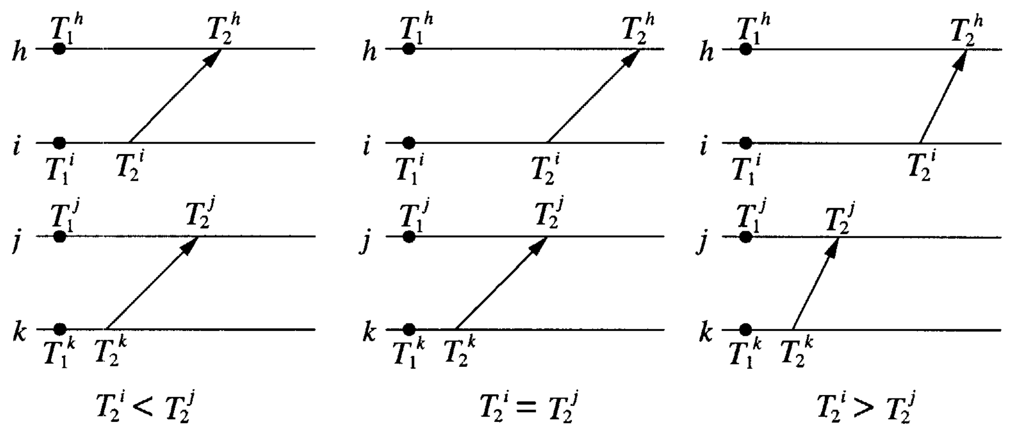


Figure 4-3: Time shifting events between independent nodes.

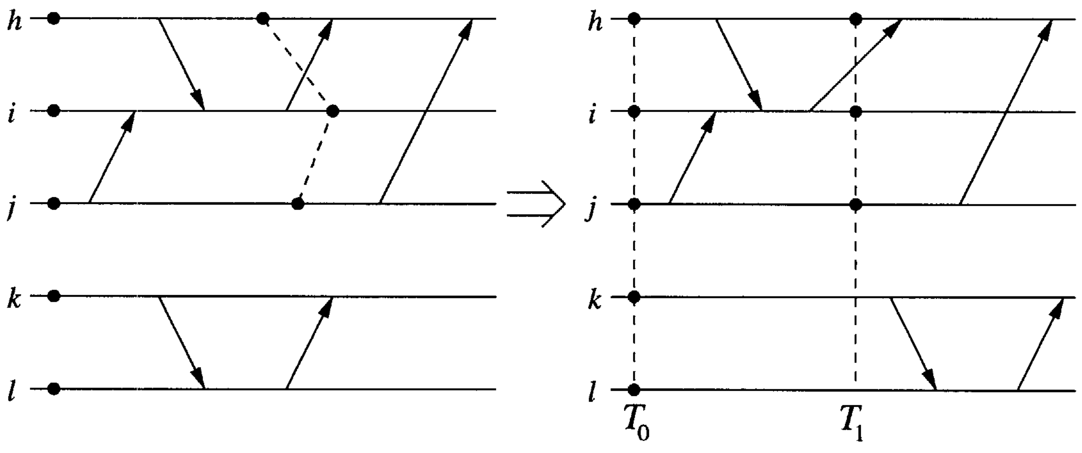


Figure 4-4: Forming a full system state from a group state.

dependencies. If two nodes i and j have not sent messages between each other since the last checkpoint that formed a global state at T_1^i and T_1^j, then T_2^i can be said to occur either before, after, or at the same time as T_2^j (Figure 4-3).

Chandy and Lamport proved that nodes in a snapshot can be time shifted to show that the checkpoints created in that snapshot can appear to be taken at the same global time. In the model in this paper, all nodes that are outside of a completed snapshot group have not sent or received messages with nodes in the group, by definition of the snapshot group. The time at the outside nodes can be shifted relative to the inside nodes. The inside nodes can be time shifted relative to each other to show that T_a^i (the local time when nodes created their checkpoint) occurred at T_1. Each outside node j that created its last checkpoint at T_b^j, can be time shifted to make all events that occurred after T_b^j occur after T_1. The outside nodes' previous checkpoint is still the current state of the node at T_1 because no events have occurred at the node. Thus, the previous checkpoints of the outside nodes and the new checkpoints of the inside nodes form a global state of the entire system at T_1. Figure 4-4 shows nodes h, i, and j in the snapshot group and k and l outside of the group. The events at k and l are time shifted to occur after T_1 and the checkpoints at h, i, and j are time shifted to occur at T_1.

4.1.2 Canceling Partial Snapshots

The previous section proved that during failure free periods, the snapshot procedure produces checkpoints that form a global state of the entire system according to the Chandy Lamport model. However, if a failure occurs during a snapshot, it may not be able to successfully complete. If the application process at a node fails after the monitor process saved the application state, then that failure will not interrupt the snapshot. If the application process fails before the monitor process saves the application state, then the snapshot must be canceled. When a snapshot is canceled, all of the nodes must discard the checkpoint they had created.

The monitor process saves the application state when it joins the snapshot and then has no further dependence on the application process during the snapshot. Because the monitor process never fails, it will continue the snapshot even if the application fails, as long as its state has been saved. The monitor process will save all messages on the inbound message links and wait for markers from other nodes and marker_sets from initiators. The local time T_a^i that the application state is saved at node i is time shifted to the global time T_1 that forms the global state of the system. Events that occur after T_a^i are not saved in this global state because they occur after the checkpoint time of the system. Therefore, an application failure after T_a^i has no impact on the snapshot. If the only failures in the snapshot group occur after the local state at the failing node has been saved, then the snapshot will successfully complete. After the snapshot completes, the failed nodes will begin the rollback procedure.

If the application process at node i fails before i receives a snapshot marker $SS_{j\rightarrow i}^k$

or in between the time when the marker is received and the application state is saved, then i is unable to create a local checkpoint. The snapshot can't complete before i joins the snapshot because initiator k will not send marker lists to the nodes before it receives the dependency list from i. If i is unable to save its application state due to a failure, then it will set the cancel flag in the message it sends to the initiator k. k will send messages to all of the nodes in the snapshot group telling them to cancel and discard their new checkpoint.

In a super-snapshot, the initiator of each snapshot will not complete its snapshot until it receives a dependencies met message from its coordinator. The coordinator will not send a dependencies met message until it receives a tentative complete message from the initiator for all of those dependencies. The initiators will only send a tentative complete message when they have received a marker set from all nodes in its group, which means that the nodes have saved their application state. Thus, when an initiator receives a dependencies met message, all of the nodes that it depends on outside of its group are guaranteed to not be interrupted by an application failure. Likewise, if a failure occurs at one node, all other nodes in the super-snapshot that depend on that node are still able to cancel their snapshot procedure and discard the new checkpoint. Similarly, after a super-snapshot has been canceled, the failing nodes will begin the rollback procedure to return the system to a previous saved state.

4.1.3 Partial Rollbacks

The rollback procedure returns the state of the system to a previous state that was saved when the system was failure free. The traditional rollback procedure restores every node to its previous checkpoint. Similar to the inside and outside groups in the snapshot process, nodes can be divided into a group that was affected by a failure and a group that was not affected. A node is affected by a failure and must be rolled back if it has received a message from an affected node or has sent a message to an affected node that was handled by the affected node before it began the rollback procedure. When a node joins the rollback group, it does not affect any further nodes because the application process is stopped and no more messages are sent or received. The

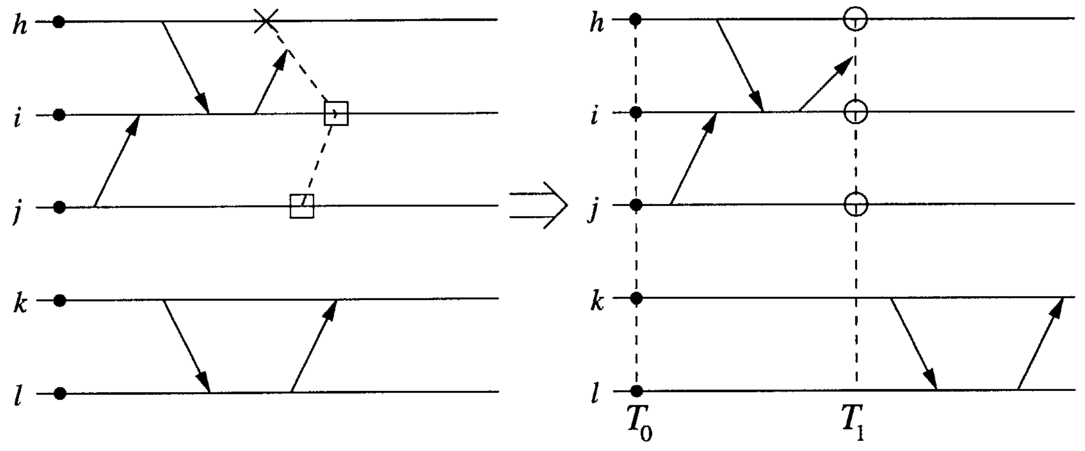


Figure 4-5: Restoring a full system state while only rolling back nodes affected by the failure.

inside and outside groups are decided using the same procedure that snapshots use. The proof that this procedure is correct was given in section 4.1.1. All inside nodes restore their last permanent checkpoint when they receive a rollback marker and then join the rollback group.

Once the inside nodes have restored, no node in the system will have application state that depends on a message that was sent or received during a rolled back period. If a node i has received a message that was sent by or sent a message that was received by a rolled back node j after j's last permanent checkpoint, then j would have sent a rollback marker, forcing i to rollback to a state before having that communication with j. The nodes outside the rollback group can be time shifted such that they have not had any events and still have the same application state that they had at their last permanent checkpoint (Figure 4-5). In the figure, h, i, and j restore their state from T_0 and remove invalid message $Msg_{i \to h}$. Nodes k and l don't need to roll back because their events have been time shifted to after T_1. They effectively have the same state at T_1 that the had at T_0. Therefore, the application process at every node is at its last checkpoint, which forms a complete global state.

In addition to returning the application processes to their previous checkpoint, the network links between nodes need to be cleared of all messages that were sent during the rollback period. The purging procedure parallels the mechanism to save messages in the snapshot procedure. All messages on an inbound link that are followed by a rollback marker are discarded before the application process resumes. Nodes don't resume the application process until they have received the marker list from the initiator and all markers in that list. Every inside node that sent a message has added the receiver to its dependency list. When rollback begins, markers are sent over all links that had messages sent over them. These markers force receiving nodes to join the rollback group and delete the messages sent during the rollback period.

When rollbacks intersect, some inside nodes join multiple rollback groups. In this case, the above proof still proves that the inside and outside groups are divided correctly and that all messages sent during a rollback period will be removed. The extension to the algorithm to handle nodes that join multiple rollbacks is that the application process does not resume until the node completes all of the rollbacks that it joined. The application must remain stopped until all rollback groups have finished to ensure that all messages sent during the rollback period are removed and do not affect the application process.

4.2 Progress

For the fault tolerance system to be effective, the snapshot and rollback process must complete in a finite time and create new checkpoints to increase the amount of saved work. This section will show that the procedures they use to coordinate groups are deadlock free. Deadlocks occur when two or more processes become stuck waiting for an event at the other process. The procedure used in snapshot and rollback can be simplified to the following actions and wait states:

1. The Initiator k in a single group
   (a) Sends markers to all nodes in $D^k_{[P,C]}$
   (b) Waits for $D^i_{[P,C]}$ until $G^k_{HJ} \supseteq G^k_{MJ}$
   (c) Sends nodes the lists of markers to expect
2. The Node i

(a) Receives the first marker $SS^k_{j \to i}$

(b) Sends markers to all nodes in $D^i_{[P,C]}$

(c) Join group and send $D^i_{[P,C]}$ to Initiator k

(d) Waits for list of markers to expect

(e) Waits for all markers in the list of markers to expect

(f) Complete and update local state

3. The Initiator k in a Super-Snapshot

(a) Sends markers to all nodes in $D^k_{[P,C]}$

(b) Waits for $D^i_{[P,C]}$ until $G^k_{HJ} \supseteq G^k_{MJ}$

(c) Reaches tentative complete state

(d) Waits for other snapshots to reach tentative complete state

(e) Sends lists of markers to expect to nodes

The three locations in a single snapshot or rollback where deadlocks can occur are 1-b, 2-d, and 2-e. As stated in the system model, all messages sent on network links will arrive in a finite time. Each node i that is sent a marker will receive it in finite time and will send out markers to all nodes in its $D^i_{[P,C]}$ if i has not already joined that group. Condition 1-b will be met in finite time because there are a finite number of nodes in the system that can perform steps 2-a through 2-c.

Once condition 1-b is met, the initiator will immediately perform step 1-c and send the lists of markers to expect to all of the nodes in the group. All nodes that are waiting at step 2-d have joined the group and will be sent one of these lists. The lists will arrive in finite time over the network and satisfy condition 2-d. Condition 2-e is also guaranteed to be met in finite time because the markers that node i is waiting for must have already been sent if condition 1-b is met. The nodes sent all of those markers in step 2-b before joining the group.

The super-snapshot process changes this procedure slightly. An additional wait is inserted after 1-b to create 3-c and 3-d. Until step 3-d, snapshots in a super-snapshot do not wait on each other. As shown above, all groups will meet the condition at step 3-b in a finite time and will reach tentative complete state. The number of groups in the system must also be finite because the system can have at most one group for each of the finite number of nodes. By the combination of these rules, all groups in a super-snapshot will meet the condition in step 3-d in finite time.

Chapter 5

Conclusion

Traditional full system snapshots and rollbacks don't scale with large systems because the cost per node grows with the size of the system. Partial system snapshots reduce the cost per node to a fixed cost based on the group size. Lowering the cost per node allows more frequent snapshots, reducing the work lost in a failure. Partial rollbacks limit the nodes that rollback to only those that have a dependency on a node that has failed. In systems that use a shared stable storage, partial snapshots can spread out the burden on the pipeline to the storage by scheduling the snapshots to take place at different times.

Determining the snapshot groups depends on the communication style of the system. Partial system snapshots can be taken as long as the communication connected set of nodes grows slowly enough that they can be snapshotted at a reasonable frequency. In a system where communication groups are determined by phases, snapshots should be performed at least once per phase. The lattice phase system described in section 2.1.2 is the worst case phase system. After a phase change and before another snapshot, this case results in the highest number of connected nodes possible. In other phase systems, groups may have common members, which would reduce the number of connected nodes after a phase change.

Partial system snapshots are more efficient, but they do introduce new complications for the snapshot process. New mechanisms are needed when sending messages during a snapshot to ensure that a message that was sent after a checkpoint isn't received before a checkpoint by another node in that group. Further, now that snapshots don't include all nodes in the system, it is possible to have multiple groups performing a snapshot at the same time. The intersections of these groups must be carefully handled to avoid deadlocks and livelocks.

The implementation given in this paper uses partial snapshots and rollbacks and addresses the above issues efficiently without being too complex. The application is allowed to continue sending messages during the snapshot by sending a marker before any messages to new nodes. Intersecting snapshot and rollback groups are combined and continued as long as a node failure doesn't cause the snapshot to be canceled.

The optimal frequency that partial snapshots should be taken is difficult to determine even with a good understanding of the distributed system and the application. The main factors in choosing a snapshot frequency are the failure rate, the cost for recovery after a failure, and the cost of snapshotting. The failure rate is generally fixed because it depends on the hardware in the system. An approximate lower bound for the snapshot frequency is typically the failure frequency because work should be able to be completed and saved in between each failure. Increasing the snapshot frequency will decrease the cost of an individual snapshot depending on the communication model of the system, but will increase the number of snapshots between failures. Likewise, if snapshots are taken more often, then the cost of recovery after a failure may decrease because fewer nodes will need to rollback. The optimal snapshot frequency is at the point where the decrease of cost for rollback equals the increase of cost for snapshots. An adaptive system could start out with an overly high snapshot frequency and then slowly decrease it while the efficiency of the system continues to increase.

The efficiency of snapshotting can be further increased if the size of the connected group of nodes is limited because both snapshots and rollbacks depend directly on the number of nodes in the group. The phase communication model for a lattice system works well because it strictly limits the number of nodes that will need to snapshot or rollback together. Any one node can not easily limit the size of the group by itself because any one node in a group can add a new node which may itself add more nodes. One approach to this issue is to have nodes send each other their list of dependencies periodically to try to determine when to snapshot before the group gets too big. The group size would not be strictly limited, but this mechanism should keep it small for the majority of snapshots. An occasional large snapshot group is allowable as long as the groups are small on average.

The system explained in section 3.3 efficiently addressed the main issues of partial system snapshots and rollbacks, but requires a fully connected network because the initiator needs to receive messages from all nodes in the group. A system similar to Koo and Toueg's algorithm (section 3.1) could operate on any network layout because each node only communicates with its immediate neighbors. The initiator of a snapshot or rollback in this system would not need to process any more data than any other node in the system. This system could allow the application to send messages during snapshot to nodes that it has already sent markers. Intersecting snapshots and rollbacks can be merged as long as a snapshot group does not depend on state from a failed node. Changes to how markers are sent and handled could also allow the network to be unreliable and not be FIFO.

Bibliography

[1] Valmir C. Barbosa and Eli Gafni. Concurrency in heavily loaded neighborhood-constrained systems. *ACM Trans. Program. Lang. Syst.*, 11(4):562–584, 1989.

[2] K. Mani Chandy and Leslie Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, 1985.

[3] M. Elnozahy, L. Alvisi, Y. M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message passing systems. Technical Report CMU-CS-96-181, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA, October 1996.

[4] Peter W. Glynn, Philip Heidelberger, Victor F. Nicola, and Perwez Shahabuddin. Efficient estimation of the mean time between failures in non-regenerative dependability models. In *WSC '93: Proceedings of the 25th conference on Winter simulation*, pages 311–316. ACM Press, 1993.

[5] David B. Johnson and Willy Zwaenepoel. Recovery in distributed systems using asynchronous message logging and checkpointing. In *PODC '88: Proceedings of the seventh annual ACM Symposium on Principles of distributed computing*, pages 171–181. ACM Press, 1988.

[6] Richard Koo and Sam Toueg. Checkpointing and rollback-recovery for distributed systems. In *ACM '86: Proceedings of 1986 ACM Fall joint computer conference*, pages 1150–1158. IEEE Computer Society Press, 1986.

[7] Sen Moriya and Tadashi Araragi. Dynamic snapshot and partial rollback algorithms for internet agent systems.

[8] A. P. Sistla and J. L. Welch. Efficient distributed recovery using message logging. In *PODC '89: Proceedings of the eighth annual ACM Symposium on Principles of distributed computing*, pages 223–238. ACM Press, 1989.

[9] Rob Strom and Shaula Yemini. Optimistic recovery in distributed systems. *ACM Trans. Comput. Syst.*, 3(3):204–226, 1985.