Chapter Four
Coordinated Checkpointing Techniques

Coordinated Checkpointing techniques checkpoint a set of processes together (the entire system or a subset thereof) in such a way that each pair of process states on stable storage are consistent with one another. The recovery algorithm is guaranteed to be able to find a recovery line and hence be able to recover a valid system state. These algorithms allow the system to be n-fault-tolerant, keep only one generation of checkpoints on disk, and avoid the domino effect.

In this chapter we present two new low-overhead algorithms based on the coordinated checkpointing of processes. The first algorithm is synchronous coordinated checkpointing of interacting sets of processes, which extends the global coordinated checkpointing technique[Tami84] to perform checkpointing and recovery of sets of interacting processes rather than of the entire system. This extension reduces the disruption to normal operation since it usually involves only a subset of processes instead of the entire system. With coordinated checkpointing of interacting sets of processes, checkpointing involves saving a consistent snapshot of the states of an interacting set of processes in such a way that a valid global checkpoint of the system state is maintained on stable storage at all times. In the proposed schemes checkpointing and recovery are done at the level of processes with no system-wide central coordination. Multiple checkpointing and recovery sessions may be active simultaneously. Unrelated sessions do not interfere with one another, while the actions of related ones are properly coordinated. Processes which are not part of the interacting set need not participate

in checkpointing/recovery and continue to do useful work. However, synchronous coordinated checkpointing, like global checkpointing[Tami84] still involves synchronous checkpoint sessions, where all members of the interacting set being checkpointed suspend normal execution for the duration of the checkpoint session.

The second algorithm is asynchronous coordinated checkpointing which achieves the benefits of checkpointing processes independently (i.e. without coordination), but without suffering from the domino effect. The key feature of this scheme is the use of volatile checkpoints in combination with the coordinated checkpointing technique to minimize disruption to normal operation due to checkpointing. A volatile checkpoint is simply a copy of the process state in local volatile memory. Checkpointing begins by copying the changed state of a process to local volatile storage, after which the process may resume execution. The rest of the checkpointing session involves identification of the set of processes to be checkpointed and the transfer of the volatile checkpoints to stable storage. Thus, a process is suspended only a small fraction of the time it takes to complete a checkpointing session. This fraction of time is independent of network delays and the number of processes that constitute the application. The cost, relative to synchronous coordinated checkpointing, is an increase in the complexity of the checkpointing algorithm which must ensure that the volatile checkpoints are saved as part of a consistent global state even though the processes being checkpointed have resumed execution. However, volatile checkpoints allow much of this local copying to be avoided through the use of a slightly modified virtual memory system. This chapter provides detailed proofs of correctness for our algorithms, based on the model presented in Chapter 2.

In the next section we provide an brief overview of previous work on coordinated checkpointing algorithms. Then we describe a global state recording algorithm, from [Chan85]. In Section 4.1.2 we show how this compares to the complete error recovery technique, global checkpointing, described in[Tami84]. We then provide high-level pseudo-code to checkpoint and roll back the states of an interacting set of processes. Several simplifying assumptions are made so that we can prove the correctness of the basic approach of checkpointing and recovering interacting sets of processes free of low-level implementation issues. These simple, short proofs are applicable to any algorithm based on checkpointing interacting sets of processes. We first show that the pseudo-code can be used to *maintain* a consistent global state on stable storage and then show that, when a failure occurs, only the interacting set(s) of the failed process(es) needs to be rolled back.

In Section 4.2 we describe how the checkpointing or rollback of an interacting set is coordinated in a tree-structured manner with the coordinator being at the dynamically-chosen root. We describe how the algorithm is designed around a two-phase-commit protocol, how independent checkpoint and recovery sessions do not interfere with each other, and how coordinator conflicts are handled. Several proofs of the correctness of these procedures are provided.

The synchronous checkpointing algorithm is described in Section 4.3 and provides details on supporting communication environments that use either virtual circuits or packet-switching. The recovery algorithm, used with either the synchronous or asynchronous checkpointing algorithms, is described in Section 4.4. Recovery from communication errors is discussed first followed by

recovery from node failures. Section 4.4.3 specifically discusses how our checkpointing algorithms are robust to (non-crash) failures.

In Section 4.5 we present the complete asynchronous coordinated checkpointing algorithm in which we minimize the interruption of normal processing due to checkpointing. In Sections 4.5.1, 4.5.2, and 4.5.3 we detail the handling of multiple ‘‘competing’’ checkpoint coordinators and checkpoint sessions. These three sections are rather similar to the synchronous algorithm, so we provide a detailing of the actions without going through lengthy proofs. Techniques for minimizing data movement during checkpointing by using an enhanced virtual memory system are outlined in Section 4.6 and then the chapter is concluded in Section 4.7. In Chapter 6 a performance analysis of these algorithms, and others, is provided.

4.1. Previous Work

Barigazzi and Strigini [Bari83] proposed an error recovery procedure for multicomputers that involves the periodic saving of the state of each process by storing it both on the node where it is executing and on another backup node. The critical feature of this procedure is that all interacting processes are checkpointed together, such that their checkpointed states are always consistent with one another. Thus the *domino effect* cannot occur and it is sufficient to store only one ‘‘generation’’ of checkpoints. The schemes presented in this chapter use this idea of checkpointing and recovering dynamically changing sets of interacting processes.

With the recovery scheme described in[Bari83] a large percentage of the

memory is used for backups rather than for active processes. The resulting increased paging activity leads to increases in the average memory access time and the load on the communication links. This load is also increased by the required acknowledgment of each message and transmission of redundant bits for error detection. The communication protocols, which are used to assure that the message ‘‘send’’ and ‘‘receive’’ operations are atomic, require additional memory and processing resources for the kernel. Thus, performance is significantly reduced relative to an identical system where no error recovery is implemented. The schemes proposed in this chapter eliminate the requirements for atomic message transmission and provide the ability to save the checkpoints on disk, where they need not have a detrimental effect on system performance. They also incorporate the use of practical, low overhead error detection mechanisms, to reduce even further the overhead required on application messages.

The idea of checkpointing and recovering interacting sets of processes is extended in[Tami84] to checkpointing and recovering the entire system (global checkpoints). That scheme does not have the disadvantages discussed above of the scheme in[Bari83]. The problem with the global checkpointing technique is that checkpointing is expensive since it requires saving the state of the entire system. Thus, for performance reasons, the time between checkpoints is relatively long (possibly tens of minutes). Hence, the system can only be used for ‘‘batch applications,’’ such as large numerical computations, where the possibility of losing minutes of computation during recovery is an acceptable price for the resulting low overhead (a few percent [Tami84]).

The global checkpointing technique has been extended to perform

checkpointing and recovery of sets of interacting processes rather than of the entire system [Koo87, Tami89]. This extension reduces the disruption to normal operation since it usually involves only a subset of processes instead of the entire system.

Our synchronous algorithm[Tami89] (described in Section 4.3 and Section 4.4) differs from the one presented in[Koo87] in several ways. First, we do not assume that the network makes channels ‘‘lossless’’, thus the definition of an interacting set is more general (a superset). We have instead specifically provided for low overhead recovery from node failures and from communication failures when the system uses either a virtual circuits or packet-switching model for communication. Also, through the use of local dynamic communication vectors we do not require any message tags (labels) on either normal or system messages. By flushing and discarding messages during a recovery session we completely avoid the livelock problem they describe. Lastly, we differ in our methods of handling interfering checkpoint/recovery sessions (see Section 4.2.1.2) and do not unnecessarily abort checkpointing sessions.

However, as in[Tami84], all of these coordinated recovery schemes still involve *synchronous* checkpoint sessions, where all members of the interacting set being checkpointed suspend normal execution for the duration of the checkpoint session. In the worst case, all processes in the system may belong to a single interacting set and all normal computation in the system has to be suspended for the entire checkpoint session. Our second algorithm, asynchronous coordinated checkpointing, requires processes to be suspended only for the taking of a local volatile checkpoint — a fraction of time that is independent of network delays and

the number of processes that constitute the application. The taking of a volatile checkpoint is highly amenable to optimization. No asynchronous coordinated checkpointing algorithm has previously been proposed.

4.2. Basic Approach of Coordinated Checkpointing

Instead of checkpointing processes independently, coordinated checkpointing techniques ‘‘simultaneously’’ checkpoint multiple processes in order to maintain a consistent global state on stable storage. *Global* checkpointing techniques record the entire (consistent) system state in every checkpointing session. Alternatively, coordinated checkpointing of interacting sets of processes can be used to update a subset of the consistent global state at a time, while ensuring the existence of a consistent global state at all times. In this section we show the correctness of both of these approaches.

4.2.1. Recording a Consistent Global State

All coordinated checkpointing techniques have in common the characteristic of keeping a consistent global state on stable storage. When a hardware failure is detected all or part of the consistent global state is used to restore a valid system state from which computation can be resumed. These techniques all share the advantage that domino effect is entirely avoided by keeping only one generation of checkpoints in stable storage.

In[Chan85] a *consistent global state* is defined as follows:

- n is the number of messages sent by process p along channel c before p’s

state is recorded,

- n' is the number of messages sent along c before c's state is recorded,
- m is the number of messages received by process q before q's state is recorded,
- and m' is the number of messages received along c before c's state is recorded.

A *consistent global state* requires $n=n'$ and $m=m'$ for all channels c between processes p and q in the system[Chan85]. If q (the receiving process) records c's state and we consider the state of channel c to actually be part of process q's state then a consistent global state requires $n=m$, for all channels c between processes p and q. In other words, instead of recording channels as separate objects, all channels are recorded as empty ($n'=m'$) and q records as part of its state all messages which have been received along c such that no messages are lost and no messages are duplicated (see Section 2.1). The initial global state is consistent and $n = 0 = m$ for all channels c between each pair of processes p and q.

To record a consistent global state Chandy and Lamport presented the following algorithm which is initiated by some process p (reworded slightly from[Chan85]):

Global State Recording Algorithm

Marker-Sending Rule for a process p:

For each channel c, incident on, and directed away from p:

— p sends one marker along c after p records its state and before p sends

further messages along c.

Marker-Receiving Rule for a process q:

On receiving a marker along a channel c:

if q has not recorded its state **then**

begin

— q records its state;

— q records the state c as the empty sequence

— q sends one marker along each channel c, incident on, and directed away from q after q records its state and before q sends further messages along c;

end

else

— q records the state of c as the sequence of messages received along c after q's state was recorded and before q received the marker along c.

If one process p initiates the global-state recording algorithm then, in order for all processes in the system to record their state a path must exist between process p and all other processes q. In other words, the system graph, where each arc is a channel and each vertex a process, must be strongly connected. Then, if no marker remains forever in an incident input channel, the global-state recording algorithm will result in each process recording its state (and the states of all incoming channels) and will terminate in finite time[Chan85].

The above algorithm assumes that channels are static for the duration of a

distributed computation. In other words, all channels that will exist between processes exist at the beginning of the computation and continue to exist until the end of the computation. Processes must also have static life for the duration of the computation.

4.2.2. Global Checkpointing

Global Checkpointing techniques[Tami84] maintain a consistent global state in stable storage by checkpointing and recovering the entire system. The algorithm presented in[Tami84] to take a global checkpoint is fundamentally quite similar to the above Global State Recording Algorithm. However, the global checkpointing algorithm was specifically designed for use with large multicomputers where the *channels* defined above are actually (directed) high-speed dedicated links and *processes* are the nodes interconnected by them. Recording the state of a node includes all local memory, any messages *flushed* from incoming links by markers (system messages), and all appropriate system tables. The target multicomputer is assumed to be strongly connected. The global state is recorded on disks which are connected to some fraction of the nodes in the system. When an error is detected, due to a hardware failure in the links or in the nodes, the entire system is rolled back to the latest global checkpoint stored on the disks. A proof given in [Chan85] shows that a (static) distributed system can be rolled back to a global checkpoint and not violate the partial ordering of events in the system.

The problem with the global checkpointing technique is that checkpointing the entire system state is expensive. Thus, for performance reasons, the time between checkpoints is relatively long (on the order of thirty minutes). This means

that the system can only be used for batch applications, such as large numerical computations, where the possibility of losing thirty minutes of computation during recovery is an acceptable price for the resulting low overhead (a few percent [Tami84]).

4.2.3. Recording Interacting Sets of Processes

The error recovery schemes presented in this chapter maintain a consistent global state on stable storage by periodically recording the states of interacting sets of processes. In contrast to global checkpointing the entire system need not suspend normal operation during checkpointing — only the interacting set being checkpointed is suspended. Asynchronous checkpointing of interacting sets (see Section 4.5) further minimizes the time that processes must be suspended. During recovery only the interacting set(s) containing the process(es) that were running on the failed node(s) are rolled back.

4.2.4. Maintaining a Consistent Global State

In this section we will prove that a consistent global state can be maintained in stable storage by recording the checkpoints of dynamically changing interacting sets of processes. In the next section we will show that interacting sets can also be used to roll the system back to a consistent state when a failure has occurred. In both sections we make several simplifying assumptions which will be addressed as implementation issues in later sections. Thus, the proofs given in the following sections are generally applicable to all error recovery techniques based on checkpointing and rollback of interacting sets of processes.

First, we briefly consider a system in which processes are *statically* partitioned into tasks such that processes within a task do not communicate with those that are outside of the task. In such a system channels need not exist between tasks. Recording the global state of this system would then require each task to record its own ‘‘global’’ state with the above Global State Recording algorithm which we might now call a Task State Recording algorithm. It is easy to see that if there are N tasks in the system, each containing some number of processes, any grouping of N recorded task states (one for each task) will form a consistent global state simply because processes within one task can not affect processes within another. In other words, there is no ordering of events in different tasks, as specified by the *happens before* relation defined in Section 2.1, and hence such events are concurrent.

We now extend this simple principle to present and prove the correctness of *updating* a recorded consistent global state by checkpointing interacting sets of processes. In this global state updating algorithm (procedure *Chkp*) subsets of system processes are *dynamically* partitioned into interacting sets of processes. In order to be able to dynamically determine the members of an interacting set of processes, each process p maintains a *Buddies* list:

Def. 4.1: For a process p, let *Buddies (p)* be the set of processes such that $q \in Buddies(p)$ if, and only if, p has either sent a message to q or received a message from q since the last checkpoint of p.

Note that in general $q \in Buddies(p)$ does <u>not</u> necessarily imply that $p \in Buddies(q)$. For example, a message may have been sent by p but not yet received by q.

We can now define an interacting set in terms of the *Buddies* list:

Def. 4.2: The *interacting set* with respect to process p_1, $Inter(p_1)$, is the set of processes containing all processes q such that either (1) $q \in Buddies(p_1)$, (2) $\exists$ a process p_2, such that $p_2 \in Inter(p_1)$ and $q \in Buddies(p_2)$, or (3) $q = p_1$.

Buddies and *Inter* can be viewed as binary relations: e.g. $q \in Buddies(p)$ as $(p,q) \in Buddies$ and $q \in Inter(p)$ as $(p,q) \in Inter$. Def. 4.2 states that *Inter* is the reflexive transitive closure of *Buddies*.

$Inter(p_1)$ is the set of processes that may be affected (directly or indirectly) by messages sent by p_1 since the last checkpoint of p_1 *or* which have produced messages which may have affected p_1 directly or indirectly. Note that, in general, $p_2 \in Inter(p_1)$ does <u>not</u> necessarily imply that $p_1 \in Inter(p_2)$.

When a process p is (periodically) checkpointed, the following procedure is performed:

Procedure Chkp (the Global State Updating Algorithm):

1) When a process p is checkpointed, all processes in $Inter(p)$ (Def. 4.2) are also marked for checkpointing.
2) All messages sent by processes in $Inter(p)$ are flushed to their final destinations.
3) The actual checkpointing is then performed.

These interacting sets are checkpointed independently to stable storage, like the above tasks, and a consistent global state (checkpoint) is always maintained. Note that if the interacting sets were static instead of dynamic the global-state updating

algorithm would be equivalent to the above task recording algorithm and, starting with an initial global state, a consistent global state would always be maintained.

Def. 4.3: Two processes, p and q, where $q \in Buddies(p)$, are said to be *buddy-consistent* if, and only if, their states reflect the same number and content of **direct** message exchanges between p and q.

Note that if $q \in Buddies(p)$ and p and q are *buddy-consistent*, then it must be the case that $p \in Buddies(q)$. This definition implies that no messages between p and q are either lost or duplicated.

Def. 4.4: A set of processes, CPS, is *consistent*, if, and only if, the following two conditions hold: (1) if $p \in CPS$ then $q \in Inter(p)$ implies $q \in CPS$, (2) for every pair of processes in CPS, $p \in CPS$ and $q \in CPS$, if $p \in Buddies(q)$ or $q \in Buddies(p)$, then p and q are *buddy-consistent*.

If the system is fault free and all messages in transit are flushed to their final destinations, all the processes in the system form a consistent process set (CPS). Note that a single process, p, for which $\{p\} = Inter(p)$, is a consistent process set.

Def. 4.5: Two processes are said to be *consistent* if, and only if, there exists a consistent process set of which both are members.

Def. 4.6: A set of processes, $DCPS$, is *dynamically consistent* if, and only if, the following holds: if the execution of all processes in $DCPS$ is blocked and all messages are flushed to their final destinations, all the members of $DCPS$ form a consistent process set.

Def. 4.7: Two processes are said to be *dynamically consistent* if, and only if, there exists a dynamically consistent process set of which both are members.

At this time we assume that the checkpoints of all processes within an interacting set are committed onto stable storage at the same ‘‘instantaneous’’ moment. Since we plan to use the consistent global state for recovery and need to recover from hardware failures which can occur at any time a consistent global state must *always* be available for recovery. In other words, the saved global state can not be allowed to be inconsistent even for a short period of time. If this requirement is imposed on Chandy and Lamport's global state recording algorithm then, when a new global state is being recorded we can not write over the previous state until the new global state has been completely saved to stable store. For example, one process p might *coordinate* the global state recording algorithm and, after all processes have written their states to stable store, issue a *commit* command which would erase the old global state[Tami84]. This same idea can be used to commit an interacting set of checkpoints. How this is achieved will be discussed in Section 4.3.

We will now prove that the above global state updating algorithm can be used to maintain a consistent global state. First we reiterate our assumptions and then introduce and prove Lemma 4.1, which holds that, when an interacting set is being checkpointed, all of the processes which are in the interacting set (and no processes not in the interacting set) record their states. Lemma 4.2 shows that when an interacting set is checkpointed, the checkpoints of every pair of processes within the interacting set are consistent (i.e. they form a *consistent process set*). Then Lemma 4.3, which follows from Lemma 4.2, shows that the checkpoints of every pair of processes within the system are always consistent (i.e. that the entire system forms a *consistent process set*).

Assumptions:

— Only one checkpoint session is ‘‘active’’ in the system at a time.

— One process p initiates the checkpoint session.

— Channels are directed, FIFO, error-free and transmit messages with arbitrary finite delay.

— Channels are static: if a distributed computation requires two processes p and q to communicate with each other then two directed channels (channel c from p to q and channel c' from q to p) exist between those processes for the duration of the computation.

— Processes are static: all processes that can fail exist at the beginning of a distributed computation and continue to exist for the duration of the computation.

— All the checkpoints of an interacting set are committed to stable storage atomically and at the same time.

— The distributed computation starts with an initial, consistent recorded global state.

— The system is fault-free.

Lemma 4.1: If $p_3 \in Inter(p_2)$ and $p_2 \in Inter(p_1)$, then $p_3 \in Inter(p_1)$.

Proof: Follows directly from the fact that *Inter* is a reflexive transitive closure of *Buddies* (see comment following Def. 4.2). □

Lemma 4.2: Assuming that the system is fault-free, the set of processes $P = Inter(p)$, which is checkpointed as described by Procedure *Chkp*, forms a *consistent process set*.

Proof: Consider $q_1 \in P$ and $q_2 \in Inter(q_1)$. By Lemma 4.1, this implies $q_2 \in Inter(p)$ so condition (1) in Def. 4.4 holds.

Assume $p_1 \in P$ and $p_2 \in P$. There are three cases to be considered: (1) $p_2 \in Buddies(p_1)$, (2) $p_1 \in Buddies(p_2)$, or (3) $p_2 \notin Buddies(p_1)$ and $p_1 \notin Buddies(p_2)$. Case (3) obviously satisfies condition (2) of Def. 4.4. The case where (1) or (2) hold is covered by the discussion of either one.

If $p_2 \in Buddies(p_1)$, since the system is fault-free and messages from p_1 and p_2 have been flushed to their final destinations, it must be the case that the state of both processes reflect the same message exchanges between them and thus they are *buddy-consistent* (Def. 4.3). Similarly, condition 2 in Def. 4.4 holds starting with $p_1 \in Buddies(p_2)$. □

Lemma 4.3: If every time a process p is checkpointed, its entire interacting set $P = Inter(p)$ is also checkpointed as described in Procedure *Chkp*, then the set of all checkpointed processes (all the processes in the system), W, forms a consistent process set.

Proof: Since W includes all the processes in the system, condition (1) in Def. 4.4 is satisfied. Consider two processes in the system checkpoint p_1 and p_2. If both processes were checkpointed together, i.e., $p_1 \in P$ and $p_2 \in P$, then, by Lemma 4.2, they satisfy condition (2) in Def. 4.4.

Assume $p_1 \in P$ and $p_2 \notin P$. If $p_2 \in Inter(p_1)$, then, based on Lemma 4.1, $p_2 \in Inter(p)$. Since this contradicts $p_2 \notin P$, it must be the case that

$p_2 \notin Inter(p_1)$. Hence, $p_2 \notin Buddies(p_1)$.

Consider the possibility that $p_1 \in Inter(p_2)$. By assumption, the p_2 checkpoint was stored prior to the p_1 checkpoint. If at the time p_2 was checkpointed it was the case that $p_1 \in Inter(p_2)$, then, following Procedure *Chkp*, p_1 was also checkpointed. Since p_2 is a checkpointed state and not the dynamic state of the process, it is guaranteed that p_2 has not sent *any* message to any other process since it was checkpointed. Hence $Buddies(p_2) = \phi$. Thus, $p_1 \notin Buddies(p_2)$.

Assume $p_1 \notin P$ and $p_2 \notin P$. Hence, both p_1 and p_2 are old checkpointed states. Thus, $Buddies(p_1) = \phi$ and $Buddies(p_2) = \phi$. Therefore, $p_1 \notin Buddies(p_2)$ and $p_2 \notin Buddies(p_1)$.

Based on the above, condition (2) of Def. 4.4 holds in all cases. □

4.2.5. Recovery From Consistent Global States

Based on Lemma 4.3, when an error occurs, we can restore a consistent system state by discarding all messages in transit and restoring all processes to their checkpointed states [Tami84]. However, in order to reduce the impact of recovery on system operation, it is desirable to reduce the number of processes that have to be rolled back [Koo87, Tami89]. When an error is detected, all the processes that *could have been* affected by the error are identified. The sets of processes that have interacted with the affected processes since their last checkpoint are determined, and the states of all these processes are rolled back to that last checkpoint. It is assumed that the entire state of a processor where an error is detected is lost. Specifically, if a processor running process p fails, we cannot obtain $Buddies(p)$ from the failed processor. The rest of this section shows

that if this procedure is followed, then, following recovery, all the processes in the system form a *dynamically consistent process set*.

The recovery operation proceeds as follows:

Procedure Rback:

1) A process p is marked for rollback.
2) Any process q, such that $p \in Buddies(q)$ is marked for rollback.
3) For each q, all processes $x \in Inter(q)$ are marked for rollback.
4) All messages in transit sent from processes marked for rollback are discarded from the network.
5) All processes marked for rollback are restored to their last checkpoint and normal operation is resumed.

Def. 4.8: If process p is marked for rollback, we denote by $Recov(p)$ the entire set of processes which must be rolled back, as determined by Procedure *Rback*.

Lemma 4.4: If the most recent checkpoints of all processes in the system form a consistent process set and Procedure *Rback* is performed for a process p, then all processes *running* on the system, R, (as opposed to their checkpointed states) form a *dynamically consistent process set*.

Proof: Since R includes all processes in the system, condition (1) in Def. 4.4 is satisfied. Consider two processes p_1 and p_2. If both processes are in $Recov(p)$, they are recovered together from their last checkpoint. Hence, by assumption, both processes are in the same consistent process set. Thus, condition (2) in Def. 4.4 holds.

Consider the case when $p_1 \notin Recov(p)$ and $p_2 \notin Recov(p)$. Assuming that the system was fault-free prior to the rollback, all processes in the system were in a dynamically consistent process set. Hence, if all messages were flushed to their destinations, condition (2) of Def. 4.4 must have held for p_1 and p_2. Since the rollback did not involve p_1 and p_2, condition (2) of Def. 4.4 must still hold.

Assume $p_1 \in Recov(p)$ and $p_2 \notin Recov(p)$.

Consider the possibility that $p_2 \in Inter(p_1)$. If p_1 was added to $Recov(p)$ in Step (2) of Procedure *Rback*, $p_2 \in Inter(p_1)$ implies $p_2 \in Recov(p)$ — a contradiction. If p_1 was added to $Recov(p)$ in Step (1) or (3) of Procedure *Rback*, *exist* $q \in Recov(p)$ such that $p_1 \in Inter(q)$. In this case $p_2 \in Inter(p_1)$ implies $p_2 \in Inter(q)$ which implies $p_2 \in Recov(p)$ — a contradiction. Hence, it must be the case that $p_2 \notin Inter(p_1)$. Note that this implies $p_2 \notin Buddies(p_1)$, therefore, if $p_1 \notin Buddies(p_2)$, then condition (2) in Def. 4.4 holds, and the proof is complete.

It remains to consider the case $p_1 \in Buddies(p_2)$. Since $p_2 \notin Recov(p)$, it must be the case that $p_2 \notin Buddies(p_1)$. Hence, the only interaction between p_2 and p_1 is a message(s) in transit from p_2 to p_1. If all messages are flushed to their destinations, the message(s) from p_2 will arrive at p_1 and $\{p_2, p_1\}$ will become *buddy-consistent*. At this point, condition (2) in Def. 4.4 will hold. □

Theorem 4.1: If Procedure *Chkp* is used for checkpointing and Procedure *Rback* is used for recovery, all processes in the system will form a *dynamically consistent process set* following recovery.

Proof: Directly from Lemma 4.3 and Lemma 4.4. □

The rest of this chapter can be viewed as describing techniques (synchronous and asynchronous) for efficient implementation of the procedures described above.

4.3. Coordinating Interacting Sets as Trees

In the previous sections we have shown how checkpointing interacting sets of processes can be used to maintain a consistent global state on stable storage and how interacting sets of processes can be recovered from that consistent global state in the event of a hardware failure. Further, in both cases the system is left in a (dynamically) consistent global state. However, we have made many unreasonable assumptions that have greatly simplified the error recovery problem. In the next three subsections we will eliminate or relax many of these assumptions. Finally, in Section 4.3 (and Section 4.4) we present a complete error recovery scheme — synchronous coordinated checkpointing.

A key problem with implementing the *Chkp* procedure is that, in order to commit the checkpoints of an interacting set atomically and at the same time, we must know when the interacting set has been found and when the states of all the processes within the interacting set (including the states of appropriate incoming channels) have been recorded to stable storage (and are therefore ready to be committed). Hence the checkpoint and recovery sessions require coordination.

To this end the process which initiates a session is chosen to coordinate that session. In reality, a handler process performs all checkpointing/recovery duties as discussed in Section 2.2. Checkpoint sessions are initiated by ‘‘timers’’ associated with each process[Bari83]. These timers may, for example, count machine instructions executed by the process. Thus, the frequency of checkpointing can be tuned to the specific needs of a task: a higher frequency of checkpointing results in higher overhead but in less work being lost when recovery is necessary. When the

processes in the interacting set begin execution again, after the checkpoint session has completed, their timers are reinitialized. Note that such timers are not an accurate method of determining when a checkpoint takes place: a process will checkpoint *at least* as frequently as is intended by the initial setting of its timer. When a checkpoint timer ‘‘goes off’’ the execution of the corresponding process, with process ID *pid*, is suspended and a *checkpoint coordinator*, with ID CC_{pid}, is started up. This checkpoint coordinator is then responsible for coordinating the checkpoint session, including committing the interacting set to disk.

When a node failure is detected a *recovery coordinator*, with ID RC_{pid} where *pid* is the process ID of a failed process, is started up in place of each failed process. Every node is assumed to maintain a list of the processes which are running on each of their neighboring node(s). Therefore, when a node fails its neighbor(s) will initiate recovery for the processes which were running on that node. In the event of multiple, simultaneous node failures a list of failed processes can always be gotten from stable storage if necessary, since every process (which might fail) must have an initial state stored on stable storage before that process begins execution. Each recovery coordinator coordinates the rollback of the interacting set to which its failed process belongs.

When an error in transmission is detected, via a signature mismatch during a checkpoint session (Section 2.4), the checkpoint handler that discovers the mismatch starts up a recovery coordinator and then aborts. The recovery coordinator is responsible for rolling back the interacting set that was in the midst of checkpointing.

In the rest of this section we address the issue of coordinating non-interfering

checkpoint/recovery sessions and then address the problems that arise when multiple interfering checkpoint/recovery sessions are allowed.

4.3.1. Coordinating the Checkpoint/Recovery Sessions

The *coordinator* handler is dynamically determined as part of each session. Using the methods described above, it is possible for more than one process within the same interacting set to initiate a checkpoint and/or recovery session. In the extreme case an interacting set with n processes may have n coordinators: i.e. if all n checkpoint timers go off ‘‘simultaneously’’, if all n processes fail, or some mixture of the two cases. In this section we assume that there is only one coordinator per session and only one session checkpointing/recovering at a time (in the system). In the next two sections we will address the issue of concurrent checkpoint/recovery sessions and multiple coordinators within a single session.

An interacting set of processes forms a *communication graph* where there is a vertex for each process and each arc indicates that communication has taken place between the two processes it connects. The communication graph can be transformed into a *communication tree* by designating one of the vertices as the ‘‘root process’’ or *coordinator*. This coordinator is the handler of the process which initiated the checkpoint/recovery session. All vertices which have arcs connected to the root (‘‘children’’ of the root) are called *first-level processes*. Processes/vertices which have no children are called *leaves*. The communication tree is the fundamental unit around which our algorithms are structured.

When checkpointing or recovery is initiated, the kernel spawns a *handler* process that performs the necessary operations. A handler initiated as a direct

result of a ‘‘checkpointing timer’’ triggering or an error being detected begins its operation assuming that it will be the coordinator of a checkpointing or recovery session. The handler forms a communication tree using the dynamic communication vector, the list of processes with whom there has been *direct* communication since the last checkpoint session (*Buddies list*), which is maintained for each process[Bari83]. This list is also called a *first-level list* since, if the process becomes a coordinator (and root of a communication tree), the processes on this list are the *first-level processes* mentioned above.

The coordinator initiates formation of a communication tree by sending *CHECKPOINT* or *ROLLBACK* messages to all the processes on its first-level list. These processes are then placed in either a ‘‘checkpointing’’ or ‘‘recovering’’ state, handler processes are spawned for them, the handlers send *CHECKPOINT/ROLLBACK* messages to all their first-level processes (except for the parent process), and so on. A process that is already part of the tree (has received a *CHECKPOINT/ROLLBACK* message) informs the sender that it will not be its child. A process is a *leaf process* if it has communicated only with processes that are already part of the communication tree. Each leaf process informs its parent that it is its child and that it is a leaf. Each non-leaf process waits for confirmations/denials from the roots of all its subtrees and then sends a confirmation acknowledgment to its parent. This level-by-level process continues back up to the root process. When the final acknowledgment is received by the root process, the communication tree is complete — the interacting set has been found.

The corresponding synchronous algorithms to establish a communication tree,

for either a checkpoint or recovery session, are as follows (*note*: these algorithms do not implement a complete error recovery scheme):

Algorithm to coordinate a recovery/checkpoint session:

A *COORDINATOR*, with ID CC_p or RC_p, is started up for process p which initiated the checkpoint/recovery session (as described above).

- If *COORDINATOR* is a checkpoint coordinator then p's state is recorded.
- Process p does not execute (sends no messages) until the session is complete.
  - *CHECKPOINT/ROLLBACK* messages are sent to each process r where $r \in Buddies(p)$.
  - The *COORDINATOR* then waits for acknowledgments (*CH_ACK/RE_ACK* messages) from all *Buddies*.
    - If the *COORDINATOR* is a checkpoint coordinator then all messages received from each *Buddy* r after p's state was recorded and before the corresponding *CH_ACK* (child or not_child) message was received from r are recorded.
    - If the *COORDINATOR* is a recovery coordinator then the sequence of messages which have been received from each *Buddy* r before the corresponding *RE_ACK* (child or not_child) message from r was received are discarded.
  - If any *CHECKPOINT/ROLLBACK* messages are received from some process q then
    - If ''checkpointing'' then record all messages received from q after p's state was recorded and before the *CHECKPOINT* message from q was received.
    - If ''recovering'' then discard any messages received from q prior to receiving the *ROLLBACK* message from q.
    - Send a *CH_ACK*(not_child)/*RE_ACK*(not_child) message to process q.
- Once *all CH_ACK/RE_ACK* messages have been received, then the communication tree has been formed and the interacting set has been found. The *COORDINATOR* sends *CH_FOUND/RE_FOUND* messages to all processes from whom a *CH_ACK*(child)/*RE_ACK*(child) message was received to inform them that the interacting set has been found.

Algorithm to participate in a recovery/checkpoint session:

A *HANDLER*, with ID CH_q or RH_q, is started up for process q which receives a *CHECKPOINT/ROLLBACK* message from some process p.

- If *HANDLER* is a checkpoint handler then q's state is recorded.
- Process q does not execute until the session is complete.
  - *CHECKPOINT/ROLLBACK* messages are sent to each process r where $r \in Buddies(p)$, except $r = p$.
  - The *HANDLER* waits for acknowledgments (*CH_ACK/RE_ACK* messages) from all *Buddies* (except $r = p$).
    - If the *HANDLER* is a checkpoint handler then all messages received from each

Buddy r after q's state was recorded and before the corresponding *CH_ACK* (child or not_child) message was received from r are recorded.

- If the *HANDLER* is a recovery handler then the sequence of messages which have been received from each *Buddy* r before the corresponding *RE_ACK* (child or not_child) message from r was received are discarded.

- If any *CHECKPOINT/ROLLBACK* messages are received from some process s then
  - If ''checkpointing'' then record all messages received from s after q's state was recorded and before the *CHECKPOINT* message from s was received.
  - If ''recovering'' then discard any messages received from s prior to receiving the *ROLLBACK* message from s.
  - Send a *CH_ACK*(not_child)/*RE_ACK*(not_child) message to process s.
- Send a *CH_ACK*(child)/*RE_ACK*(child) message to process p and wait for a *CH_FOUND/RE_FOUND* message from p.

If we carefully examine procedure *Chkp* and the above algorithms we see that, for a checkpoint session, the above *COORDINATOR/HANDLER* algorithms clearly implement the *Chkp* procedure. These algorithms are similar to the original Global State Recording Algorithm, where *CHECKPOINT* and *CH_ACK* messages serve the role of markers. The *COORDINATOR* algorithm includes both the marker-sending and marker-receiving rules while the *HANDLER* algorithm implements just the marker-receiving rule.

4.3.1.1. Multiple Independent Checkpoint/Recovery Sessions

The previous sections did not handle the existence of more than one checkpointing or recovery coordinator running at the same time and/or more than one active checkpointing or recovery session. However, in our system model, any process can initiate a checkpoint/recovery session (and attempt to be that session's coordinator) at any point in time. Checkpoint/recovery messages have the identifier of the coordinator associated with them. This identifier consists of the coordinator's node ID and process ID (for the entire system the (node ID, process

ID) pairs are totally ordered). When a process receives a checkpoint/recovery message, it becomes a member of the interacting set identified by that coordinator ID (a member of $Inter(z)$ for coordinator z). In the following two sections, we present strategies for handling the situation of a process receiving additional checkpoint/recovery messages with different coordinator IDs after having already joined an interacting set.

After receiving the first *CHECKPOINT* or *ROLLBACK* message, specifying CC_y or RC_y, respectively, as the coordinator, a process p may receive from some process q a second *CHECKPOINT* or *ROLLBACK* message specifying a coordinator CC_z or RC_z, where $z \neq y$. There are two possible cases to be considered:

1) Following the procedures for forming the communication tree described earlier, the tree coordinated by y will be formed (y will receive all the *CH_ACK/RE_ACK* messages) *without* including q.

2) Following the procedures for forming the communication tree, the tree can not be formed without including q.

The first case means that once the tree is formed $q \notin Inter(y)$. The second case means that once the tree is formed $q \in Inter(y)$. For the rest of this section we will consider the first case; we will address the correct handling of competing coordinators (case 2) in the next section.

Lemma 4.5: Consider a process p, which is already participating in a checkpoint/recovery session after receiving a *CHECKPOINT* or *ROLLBACK* message specifying the coordinator to be CC_y or RC_y, respectively. If p receives from a process q a *CHECKPOINT/ROLLBACK* message specifying

a different coordinator, CC_z or RC_z, then $q \notin Buddies(p)$ and $p \in Buddies(q)$.

Proof —

The procedures for checkpointing and recovery specify that the only way p can receive a *CHECKPOINT/ROLLBACK* message from q is if $p \in Buddies(q)$. Since the first *CHECKPOINT/ROLLBACK* message p receives specifies y as the coordinator, it must be the case that $p \in Inter(y)$. By the assumption for this section, when p received this *CHECKPOINT/ROLLBACK* message and until the time that the tree formation completed (y received all the *CH_ACK/RE_ACK* messages) it was the case that $q \notin Inter(y)$. If, at any time prior to or during the formation of the tree it was the case that $q \in Buddies(p)$ then, by Def. 4.2, it would also be the case that $q \in Inter(y)$, violating the assumption for this section. Thus the only way it would be possible for q to be a member of $Buddies(p)$ is if p sent or received a normal message from q *after* y completed forming its communication tree. However as discussed in Section 4.2.1, once the communication tree is formed, participants in the session do not send or receive *any* normal message. Hence it is not possible for q to become a member of $Buddies(p)$ until the checkpoint/recovery session coordinated by y is completed. Thus $q \notin Buddies(p)$. □

In other words, this case occurs only when a process q sends a message to p which arrives during p's checkpoint/recovery session (*after* the session has begun) and *before* q joins/starts a checkpoint/recovery session. Another way of stating the above is that q is not in p's interacting set but p is in q's interacting set, from the perspective of p and q, respectively. In the next section we deal with the case

where q might actually be in p's interacting set (via *indirect* communication with p) but *looks* like it is in a different interacting set because it is sending a *CHECKPOINT/ROLLBACK* message with a different coordinator ID.

Now, there are four permutations that need to be addressed: when a process p is in the midst of a checkpoint or recovery session a *CHECKPOINT* or *ROLLBACK* message, *with a different coordinator ID*, may arrive from some process q. We now elaborate on how these cases are handled and motivate their correctness by showing that $n = m$ is preserved (no messages are lost or duplicated) for the checkpoints, or recovered states, of p and q in each of the four cases (Theorem 4.2): We define the sequence of events that comprise a failed process q to be: $e_0^q \leq e_{c-1}^q \leq e_c^q \leq e_f^q$, where

(1) e_0^q is the first event which occurs while q is in its initial state,

(2) e_{c-1}^q is the last event to occur in q before q took its (last) checkpoint,

(3) e_c^q is the first event that occurred in q after q took its checkpoint, and

(4) e_f^q is the last event that occurred in q before q failed.

The events of a non-failed process p can be defined similarly simply without state e_f^p.

case 1: If p is involved in a checkpoint session and receives a *ROLLBACK* message from q then we could:

— abort the current checkpoint session and roll back p's interacting set (as well as q's interacting set) or

— complete p's checkpoint session and discard all messages which have arrived from q prior to receiving the *ROLLBACK* message.

Clearly the latter method is preferred, so as to minimize work that is lost during recovery.

Proof —

Recall, that for this section, and thus all four cases discussed here, a different coordinator ID on a *CHECKPOINT* or *ROLLBACK* message indicates that the sending and receiving processes are not part of the same interacting set.

Since p is in the midst of a checkpoint session (and is therefore suspended from executing), there is no event $e_j^p, e_c^p \leq e_j^p$ which is causally dependent on some event $e_i^q, e_c^q \leq e_i^q \leq e_f^q$. Further, there is no event $e_i^q, e_c^q \leq e_i^q \leq e_f^q$ which is causally dependent on some event $e_j^p, e_j^p \leq e_c^p$ (or upon which e_j^p is causally dependent) (From Lemma 4.5). By discarding the message(s) m that q sent to p (which resulted in $p \in Buddies(q)$, but which have not yet been processed by p) we remove any possible dependency between p and q. Therefore p does not have to roll back and $n = m$ between the checkpoints of p and q when p's checkpoint session completes (From Lemma 4.3). □

case 2: If p is involved in a recovery session and receives a *ROLLBACK* message from q then all messages that arrived from q prior to receiving the *ROLLBACK* message must be discarded and p simply continues its own recovery sessions and sends a *RE_ACK(not child)* message to q.

Proof —

Since the checkpoints of processes p and q are consistent (Lemma 4.3) then when p and q are rolled back, and all channels between them are cleared of messages, then $n = m$ (Follows from Lemma 4.4). □

case 3: If p is involved in a recovery session and receives a *CHECKPOINT*

message from q then p must either:

— cause q to switch from a checkpoint session to a recovery session, or

— join q's checkpoint session *after* p's recovery session has completed.

The latter method is preferred in order to minimize work that is lost when a failure occurs.

Proof —

The ''old'' checkpoint of q and p's checkpoint are consistent (Lemma 4.3). There exists no event $e_j^p, e_c^p \leq e_j^p$ that causally depends on any event $e_i^q, e_c^q \leq e_i^q$ (and vice versa), (From Lemma 4.5). Therefore p can roll back to its checkpoint and q does *not* have to roll back (extends from from Lemma 4.4). Process q can not be committed to its current checkpoint until after p replies (sends a *CH_ACK*) to q (from the algorithms above). Process p does not reply to q until its checkpoint session has been committed (also from algorithm). However, p's checkpoint will not be consistent with q's new checkpoint unless p checkpoints as part of q's interacting set. Since $p \in Buddies(q)$ p receives a *CHECKPOINT* message from q and, as a result, joins q's checkpoint session. At the end of the checkpoint session the checkpoints of p and q will be consistent (Lemma 4.2). Therefore, given the assumption that the checkpoints of an interacting set are committed atomically and at the same time, then at no time are the checkpoints of p and q not consistent. □

case 4: if p is involved in a checkpoint session and receives a *CHECKPOINT* message from q then p could either:

— join q's checkpoint session *after* completing its own checkpoint session, or

— abort q's checkpoint session — i.e. it may be desirable to adopt a system policy that prevents any process from participating in more than one checkpoint session at a time.

Proof —

First we examine the approach where p joins q's checkpoint session after completing its own checkpoint session.

At the end of p's checkpoint session p's new checkpoint will be consistent with q's old checkpoint (Lemma 4.3). Process q will not be committed to its new checkpoint until p replies (sends a *CH_ACK*) to q (from the algorithms above). Process p does not reply to q until its checkpoint session has been committed (from algorithms). To be consistent with q's new checkpoint p must checkpoint as part of q's interacting set, which it will do since $p \in Buddies(q)$. At the end of q's checkpoint the checkpoints of p and q will be consistent (Lemma 4.3). Therefore, given the assumption that the checkpoints of an interacting set are committed atomically and at the same time, then at no time are the checkpoints of p and q inconsistent. □

Second, p aborts q's checkpoint session.

In order to abort a checkpoint session we need to introduce a new system message *ABORT* which p would send to q. Process q would then forward this message to all processes to whom it previously sent a *CHECKPOINT* message *or* received a *CHECKPOINT* message from (and so on). Then q would cease all checkpoint duties and destroy the new checkpoint. Eventually all processes in q's interacting set would receive the *ABORT* message (proof follows trivially from Lemma 4.1). Since q's checkpoint can not be committed until p replies to q with a *CH_ACK*

message (from the algorithms), then q's checkpoint will clearly never be committed if p replies with an *ABORT* message. Process q's ''old'' checkpoint (only checkpoint after the new one is destroyed) is consistent with p's new checkpoint (from Lemma 4.3). Further, if p's checkpoint is also aborted (i.e. by some process in q's interacting set), the ''old'' checkpoint of p is consistent with the ''old'' checkpoint of q by the same argument. □

For the remainder of this thesis we will assume that the first option is chosen: in other words we will not allow a process which belongs to one checkpoint session to abort another checkpoint session, since, in the worst case ''deadlock'' could occur where processes p and q never checkpoint. We have now shown that, for all four cases, $n = m$ is preserved for the checkpoints, and recovered states, for two processes p and q where p and q belong to different interacting sets (and therefore different checkpoint/recovery sessions). This completes our informal proof.

4.3.1.2. Multiple Coordinators

In the previous section we assumed that two system messages that contain different coordinator IDs were sent by processes belonging to two different interacting sets. Therefore, when a process p involved in a session coordinated by a process x receives a *ROLLBACK/CHECKPOINT* message from some process q with coordinator y, p and q are guaranteed to be in distinct interacting sets. In this section we relax this assumption completely. Since it is possible for several processes *within a single interacting set* to initiate checkpointing and/or recovery sessions ''simultaneously'', processes p and q *might* actually be in the *same* interacting set, in which both x and y initiated a checkpoint/recovery session. In

order to be able to commit the checkpoints of an interacting set atomically and at the same time we must have a single coordinator for each session. To this end we *deterministically* choose a checkpoint/recovery coordinator from amongst those processes that initiated a checkpoint/recovery session.

One simple protocol is to always choose recovery over checkpointing, when a *ROLLBACK* message is received by a checkpointing process, and to always choose a ‘‘larger’’ coordinator ID over a ‘‘smaller’’ coordinator ID, when a *ROLLBACK* message (with a different coordinator ID) is received by a recovering process or a *CHECKPOINT* message (with a different coordinator ID) is received by a checkpointing process[Tami89]. Due to the stepwise confirmation/denial process of the algorithms it is possible to create a correct a single communication tree with one coordinator by ‘‘disassembling’’ all but one of the subtrees and incorporating their members in the single ‘‘winning’’ tree. If a checkpointing process receives a *ROLLBACK* message, the *ROLLBACK* message ‘‘wins’’ regardless of the coordinator ID. If a checkpointing (or recovering) process receives another *CHECKPOINT* (or *ROLLBACK*) message with a larger ID, the new message ‘‘wins’’. In any case the process propagates the winning *CHECKPOINT* or *ROLLBACK* messages to all its first-level processes, regardless of whether or not it previously propagated any losing *CHECKPOINT* or *ROLLBACK* messages.

Eventually, the step-by-step propagation of the winning session ‘‘flushes out’’ all remnants of the losing session and a consistent communication tree for checkpointing or recovery is established. The proof of this follows easily from Lemma 4.1. Since the decision is deterministic all handlers will make the same decision and all processes in the interacting set will receive a message propagating

the new decision.

While this method is fairly simple, we actually roll back more processes than is absolutely necessary — as will shortly become apparent. To get a better protocol we turn back to the proof from the previous section (Theorem 4.2). Now that more than one process can initiate a session we still have the same four cases as before, except that we can no longer assume that process q belongs to a different interacting set than p.

We examine the four cases again, where multiple checkpoint/recovery sessions may coexist in the system at the same time *and* any process has the ability to initiate a checkpoint/recovery session at any time.

Rules to handle conflicting coordinator IDs and/or sessions:

case 1: A checkpointing process p with coordinator CC_x (where p may equal x) receives a *ROLLBACK* message from a process q with:

a) a different coordinator ID CC_y, or *b)* the same coordinator ID.

case a: a different coordinator ID, and

— $q \in Buddies(p)$

Process p switches to a recovery session with coordinator CC_y and will send a *RE_ACK(child)* message to q. Switching to a recovery session requires propagating the decision (sending *ROLLBACK* messages with proper *CCID*) to all processes to whom p sent previously sent *CHECKPOINT* messages, and to the (parent) process from whom p received a *CHECKPOINT* message if $p \neq x$.

— $q \notin Buddies(p)$

Process p continues the checkpoint session.

If CC_x receives all expected *CH_ACK* messages then p will receive a *CH_FOUND* message. Process p will then send a *RE_ACK(not child)* message to q.

Otherwise, if CC_x receives a *ROLLBACK* message from a process $r \in Buddies(p)$ (case a) then p will switch to a recovery session and send a *RE_ACK(child)* message to q.

case b: with the same coordinator ID,

— Process p switches to a recovery session (with coordinator CC_x), propagates the decision, and sends a *RE_ACK(child)* message to q.

case 2: A recovering process p with coordinator CC_x receives a *ROLLBACK* message from a process q with:

a) a different coordinator ID CC_y, or *b)* the same coordinator ID.

case a: with a different coordinator ID, and

— $q \in Buddies(p)$

Process p chooses the larger coordinator ID, x or y. If CC_y ''wins'' then the switch from CC_x to CC_y is propagated to all of p's *Buddies* (except for q) via *ROLLBACK* messages tagged with CC_y. A *RE_ACK(child)* message is sent to q. If CC_x ''wins'' ID then q will initiate the coordinator switch when it receives the *ROLLBACK* message p sent.

— $q \notin Buddies(p)$

Process p simply continues with its own recovery session.

Process p will later send a *RE_ACK(not child)* message to q when it receives a *RE_FOUND* message, or, if p later receives a *ROLLBACK* message with coordinator CC_y (and $y > x$) from one of p's *Buddies* then p will switch to coordinator CC_y and send a *RE_ACK(not child)* message to q.

case b: with the same coordinator ID,

Process p simply discards all messages received from q prior to receiving the *ROLLBACK* message, sends a *RE_ACK(not child)* to q, and continues with its recovery.

case 3: A recovering process p with coordinator CC_x receives a *CHECKPOINT* message from a process q with:

a) a different coordinator ID CC_y, or *b)* the same coordinator ID.

case a: with a different coordinator ID, and

— $q \in Buddies(p)$

Process p ignores the *CHECKPOINT* message.

— $q \notin Buddies(p)$

Process p will join the checkpoint session after the recovery session is done, unless q later sends a *ROLLBACK* message — in which case, if p is still recovering then case 2, otherwise if p has resumed executing then p will rollback with q.

case b: with the same coordinator ID,

Process p ignores the *CHECKPOINT* message.

case 4: A checkpointing process p with coordinator CC_x receives a *CHECKPOINT* message from a process q with:

a) a different coordinator ID CC_y, or *b)* the same coordinator ID.

case a: with a different coordinator ID, and

— $q \in Buddies(p)$

Process p chooses the larger coordinator ID, x or y. If CC_y ''wins'' then the switch from CC_x to CC_y is propagated to all of p's *Buddies* (except for q) via *CHECKPOINT* messages tagged with CC_y. A *CH_ACK(child)* message is sent to q. If CC_x ''wins'' ID then q will initiate the coordinator switch when it receives the *CHECKPOINT* message p sent.

— $q \notin Buddies(p)$

Process p simply continues with its own checkpoint session.

Process p will later join q's session as q's child after p finishes its current checkpoint session unless q and p's interacting sets later merge (e.g. if p later receives a *CHECKPOINT* message with coordinator CC_y (and $y > x$) from a process $r \in Buddies(p)$ then p will switch to coordinator CC_y and send a *CH_ACK(not child)* message to q).

case b: with the same coordinator ID,

Process p records all messages received from q prior to receiving the *CHECKPOINT* message, sends a *CH_ACK(not child)* to q, and

continues with its checkpoint session.

In general, if processes p and q belong to sessions with the *same* coordinator ID, then p and q belong in the same interacting set. In such a case, a checkpointing process p receiving a *ROLLBACK* message from q (with the same coordinator ID) will switch over to a recovery session; a recovering process p receiving a *ROLLBACK* message from q will discard all messages previously received from q and send a *RE_ACK(not child)* message to q; a recovering process p receiving a *CHECKPOINT* message ignores that message (since q will switch over to a recovery session); and a checkpointing process p receiving a *CHECKPOINT* message saves all messages previously received from q and sends a *CH_ACK(not child)* message to q.

If processes p and q belong to sessions with *different* coordinator IDs, then it must be determined whether or not p and q belong to the same interacting set. If process p receives a *ROLLBACK/CHECKPOINT* message from q and q is in p's view of the interacting set ($q \in Buddies(p)$) then p and q clearly belong to the same interacting set. Thus, if p and q belong to the same type of session, then the larger coordinator ID is chosen and p or q (whichever one belonged to the losing coordinator's session) takes steps to flush out the smaller coordinator ID by resending the *ROLLBACK/CHECKPOINT* messages with the winning coordinator ID. If p and q are in different types of sessions then the recovery session with ''win'' out over the checkpoint session.

Otherwise, if q is *not* in p's view of the interacting set ($q \notin Buddies(p)$) then p continues with its checkpoint/recovery session. If some other process r which is in q's interacting set (where r could be q) and some process s which is in p's

interacting set (where s could be p, but *NOT* ($s=p$ *AND* $r=q$)) are in each others view of the interacting set then processes s and r will cause the two interacting sets to merge into one. However, if there are no such processes s and r then p will eventually receive the *CH_FOUND/RE_FOUND* message that it expects and, knowing that p and q are in separate sessions, will send a not child acknowledgment to q (or vice versa) and simply continues with the session. Note: ''will eventually receive the *CH_FOUND/RE_FOUND*'' will happen only if deadlock does not occur. This deadlock problem (and our solution) is discussed in Section 4.5.3.

This protocol may be somewhat difficult to comprehend because of the subtle difference between one process being in another process' *view* of the interacting set and two processes actually being in the same interacting set. The complexity stems from the existence of the following situation: a process p is in process q's view of its (q's) interacting set ($p \in Buddies(q)$) while q is not in p's view of its (p's) interacting set ($q \notin Buddies(p)$) and we may later decide that p and q are *not* in the same interacting set. The important point to note is that we *cannot* simplify the problem by saying ''if one process is in the view of another then the two processes are in the same interacting set''.

It is easiest to illustrate this through an example. Assume we have the aforementioned situation and that p and q are both in the midst of checkpointing. Further there exists no processes r and s (from p and q's interacting sets, respectively) which are both in each others view of the interacting set. In our protocol the interacting sets never merge — p's checkpoint session completes and then p joins q's checkpoint session. Why can't the checkpoint sessions of p and q

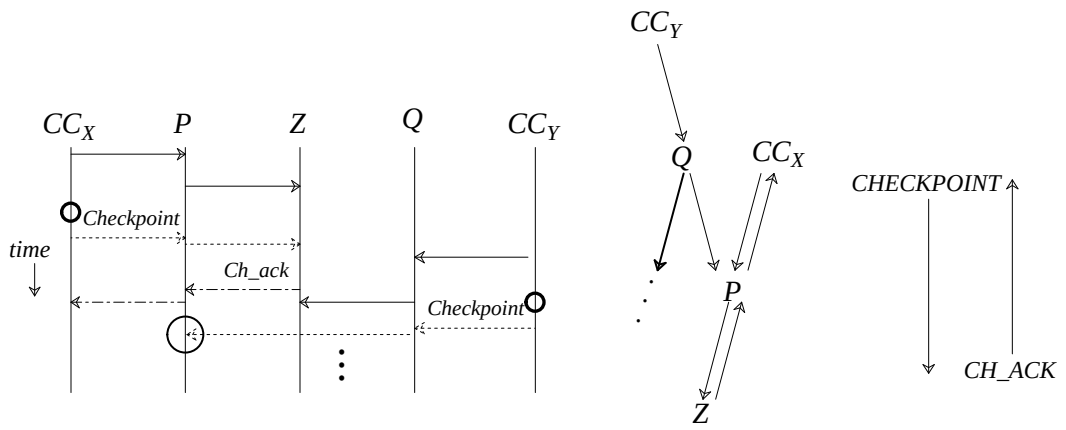

Figure 4.1: Should process p join q's checkpoint session, after its own session has finished, or should the two sessions merge into one? This figure is an example of why two sessions should not always be merged. If the sessions do merge then CC_x has no way of knowing when the interacting set is complete.

simply always be merged instead of p later joining q's session? The reason is as follows — in Figure 4.1 p has received *CH_ACK* messages from its only child z and has sent a *CH_ACK* message to its parent (and coordinator) x. Recall that q is not in process p's view of the interacting set, hence p did not send a *CHECKPOINT* message to q and does not wait for a *CH_ACK* message from q. At the time that x receives the *CH_ACK* message from its only child it expects that the interacting set is now complete. If p were to allow q's interacting set to merge with its own at this time then p has expanded its interacting set from processes p, x, and z to include q and all the processes with which q has communicated. The coordinator, x, must be informed of this expansion because the expanded interacting set is not complete since q has not finished receiving *CH_ACK* messages from its children. Hence x does not know when the interacting set is finally complete and therefore checkpointing will not complete. To this end, the *CH_ACK* message must be a guarantee that no more *new* processes will join the

interacting set hence the problem is solved by not allowing q to join p's checkpoint session - instead p will later join q's session.

4.4. Synchronous Coordinated Checkpointing

In the previous section we have shown how checkpointing and recovery of interacting sets of processes can be coordinated in a distributed manner, where any process can coordinate (or attempt to coordinate) a checkpoint/recovery session at any time, and a checkpoint/recovery session can be initiated at any time. With the technique presented, checkpoint/recovery sessions will be merged into a single session, with all but one coordinator eliminated. Further, any number of checkpoint and recovery sessions can be active in the system at a time.

In this section the complete synchronous checkpointing scheme is presented, including a detailed communication model whose use relaxes the assumption made in section 4.1.4 that channels are directed, FIFO, error-free, static, and transmit messages with arbitrary finite delay. In this section and the section on recovery (Section 4.4) the technique by which the checkpoints (of an interacting set of processes) are atomically committed to stable storage will be detailed. The resulting scheme is then robust to the ‘‘simultaneous’’ failure of an arbitrary number of processes.

4.4.1. Supporting Dynamic Virtual Channels and Processes

In this section a simple model for communication between processes using virtual channels is presented. This model employs the error detection methods discussed in section 2.4, and is, as a result, able to provide directed, FIFO, error-free channels, which can be dynamically created and destroyed, at very low overhead. The communication between processes using virtual channels can be modeled as follows:

- Each process p has some number of *open* unidirectional virtual channels of which p is either source or destination. We assume that source and destination addresses include the unique address of the node on which the process currently resides.
- Each end of a virtual channel has a FIFO message queue and a *signature* associated with it. This signature can be stored in a linear feedback shift register as discussed in Section 2.4. We denote all outgoing queues as $SendQ_p[q]$ with signature $Ssig_p[q]$, where p and q are the source and destination processes of the channel, respectively. All incoming queues are denoted $RecvQ_p[q]$ with signature $Rsig_p[q]$, where p and q are the destination and source processes of the channel, respectively.
- A *dirty bit* is associated with each end of the virtual channel. The dirty bit is reset at the end of a checkpoint/recovery session and set when a message is sent/received on the channel.
- When the channel is first opened, via a special ‘‘create circuit’’ message routed hop-by-hop from the source process to the destination process, the two signatures at the ends of the channel must be initialized to the same value, the

channel must be empty, and the *SendQ* and *RecvQ* associated with the channel must also be empty.

- When a channel is closed a special ‘‘destroy circuit’’ message is sent down the channel (from source to destination) which flushes any messages in transit on the channel to the destination process. This message carries the source signature of the channel which is compared to the destination signature of the channel. If a mismatch is found then recovery is initiated (see Section 4.4). Otherwise, if the channel has been used since the last checkpoint session involving the source and destination processes (i.e. the dirty bits are set), then the channel is put on a list of *closed* channels at both the source and destination processes. Thus the *dynamic communication information* (*Buddies* list) of a process is determined from all the dirty open and closed channels it has. The list of closed channels associated with a process is destroyed at the end of that process' next checkpoint.
- When a process p executes a $SEND(M, dst)$ the message M is appended to $SendQ_p[dst]$. When a process p executes a $RECEIVE(M, dst)$ the message M is removed from the head of $RecvQ_p[src]$. A process can receive a message from a specific destination or on a first-come-first-served basis and may send a specific message to more than one process.
- A MsgHandler, one per node, is responsible for handling the actual sending and receiving of messages at a node. The MsgHandler periodically sends messages in some ‘‘fair’’ way by removing a message M from the head of some $SendQ_{src}[dst]$, updating the corresponding $Ssig_{src}[dst]$, and sending the message out on the link. When a message M arrives at a node MsgHandler

appends M to the appropriate $RecvQ_{dst}[src]$ and updates $Rsig_{dst}[src]$. The use of LFSRs (linear feedback shift registers) allows the updating of *Rsig* and *Ssig* to take place in parallel with message receipt and transmission. If the arriving message is a *fail-safe* message (i.e. a recovery or checkpointing message, see Section 2.4), then, after it is determined to be free of errors, it is delivered to the appropriate checkpoint or recovery handler.

- The checkpoint or recovery handler of some process p sends its fail-safe messages to the handler of some process q by appending them to $SendQ_p[q]$. Any fail-safe messages, when sent (or received) off-node by MsgHandler, are not used to update the signature(s) associated with that channel. Instead MsgHandler either examines or appends error correcting codes onto the fail-safe messages before delivering or sending them, respectively.

In Section 4.1 we assumed that all communication took place over static channels that guaranteed delivery of error-free messages in FIFO order with arbitrary finite delay. The model we have now adopted is more realistic. Channels between processes are created and destroyed dynamically and at any time. However, when a channel is destroyed sufficient information is kept around until the next checkpoint session involving the processes on either end of the channel. We have presented a low overhead error detection mechanism which can be used for communication between application processes. Lost messages, including those which were in transit through a node when that node fails, are detected during the next checkpoint session involving the sender and intended receiver processes. Recovery is then initiated at that time (see Section 4.4).

The same model can be used in a system employing *packet-switching*, instead

of virtual channels, with a couple changes. Application messages are routed hop-by-hop from source to destination, hence no ‘‘create/destroy circuit’’ messages are used. Also, instead of using dirty bits, a *list* of processes with whom direct communication has taken place must be maintained between checkpoints.

As previously discussed, every process that can fail must have an initial state stored on stable storage before execution can begin. Furthermore, every process must take a checkpoint upon termination. This is necessary to ensure that the process has executed correctly, since immediate error detection is not performed. It is also necessary in order to ensure that other processes, which have communicated with the terminating processes since its last checkpoint (i.e. processes in the terminating processes interacting set), will have a checkpoint to which they can correctly roll back after that process has terminated. Once this checkpoint has taken place, then all traces of the process including the checkpoint just taken can be erased. As an optimization the checkpoint of a terminating process need not actually be physically sent to disk. Instead a ‘‘dummy entry’’ can be maintained by the disk server for the new checkpoint. When the dummy entry becomes ‘‘valid’’ all entries relating to the corresponding process are removed.

4.4.2. Synchronous Checkpointing of Interacting Sets

Checkpointing is triggered by a ‘‘checkpointing timer,’’ which causes the kernel to spawn a *handler* process. This handler begins its operation assuming that it will *coordinate* a checkpointing session for the interacting set of the process with which the timer is associated. In synchronous checkpointing, once a process becomes involved in a checkpoint session it does not execute again until its

checkpoint has been committed to disk. Checkpointing each process involves the following basic steps:

1. The process is halted. A *handler* process, that will participate in the checkpointing session on behalf of the halted process, is initiated.
2. The portion of the process state that is not in memory (e.g. contents of registers) is stored in a dedicated buffer area in local memory.
3. The process state in memory (or, as discussed in Section 4.6, just the modified pages) is sent to stable storage.
4. The interacting set/checkpoint tree is found. The communication channels between processes in the interacting set are *flushed* and any packets flushed from incoming channels are sent to stable storage as part of the checkpoint state.
5. Once all the checkpoint states, including flushed packets, of all the processes in the interacting set have been written to stable store, those states are committed and the handlers terminate. Processes then resume normal operation.

If the checkpointing session is triggered by the timer of process X, the *checkpoint coordinator*, CC_X, is responsible for sending process X's state to disk, finding the current interacting set (*checkpoint tree*) by sending *CHECKPOINT* messages (markers with error detections bits) to all *first-level processes* Y (as defined in Section 4.2.1), and checking for communication errors on all channels from processes Y to process X (as described in Sections 2.4 and 4.3.1).

Any process, Y, receiving a *CHECKPOINT* message will be suspended and a

checkpoint handler (CH_Y) will be started up in its place. This handler is responsible for copying Y's state and sending it to disk; sending *CHECKPOINT* messages to all first-level processes *except* for the coordinator/parent; and checking for any errors on all of Y's incoming channels (virtual circuits). This continues until *leaf processes* are found — processes which have communicated only with processes already a part of the tree. The checkpoint handler for each leaf process then sends a *CH_ACK* message (marker with error detections bits) to its ''parent''. Any handler receiving a second, or more, *CHECKPOINT* message (i.e. already has a ''parent'' in the checkpoint tree) receives and sends a marker and checks for communication errors but denies child status of that handler.

Once a parent of a leaf process receives *CH_ACK* messages from all of it's children, a *CH_ACK* message is sent to its parent, and so on. When CC_X receives *CH_ACK* messages from all processes to which it sent *CHECKPOINT* messages, the entire interacting set has been found. Thus, *CHECKPOINT* and *CH_ACK* messages serve the purpose of *flushing* messages to their destinations, carrying error detection bits from one end of a channel to the other for comparison, and finding the interacting set.

Once CC_X receives *CH_ACK* messages from all its first-level processes, it sends a *CH_FOUND* message down the tree, thus notifying the handlers that all messages have been flushed to their destinations and that message queues can now be sent to disk. Once these queues have been saved, *CH_DONE* messages are sent from the leaf processes to their parents, and so on. When CC_X receives *CH_DONE* messages from all its children, it ''knows'' that the checkpoints of all processes in the tree have been written to disk.

CC_X commits all processes to the new checkpoint by directing its disk node to commit to the new checkpoint of X and to destroy X's previous checkpoint. When the disk node acknowledges this operation, CC_X sends *CH_COMMIT* messages down the tree. *CH_RESUME* messages are then sent up the tree after each handler commits its process' state to disk. Each handler which sends a *CH_RESUME* flags its process as ''runnable'', clears the *first-level list* for its process (which contains the dynamic communication information — the *Buddies* list) and terminates. CC_X terminates last. Processes cannot participate in or initiate new checkpointing sessions until they receive the *CH_RESUME* message.

4.4.3. Concurrent Invocations of the Algorithms

As discussed in Section 4.2, it is possible for several processes within an interacting set to initiate checkpointing and/or recovery sessions ''simultaneously''. To solve this problem a single coordinator is deterministically chosen to coordinate the session. To this end, *CHECKPOINT* and *ROLLBACK* messages contain checkpoint/recovery coordinator IDs and all handlers locally store their current coordinator ID. Since there is a total ordering of node and process identifiers, a process receiving *CHECKPOINT* or *ROLLBACK* messages originating from different coordinators can pick the coordinator with the ''largest'' ID.

As discussed in Section 4.2.1.2 a simple protocol would be to always choose recovery over checkpointing: i.e. *ROLLBACK* messages always win over *CHECKPOINT* messages regardless of the coordinator ID. Specifically, a *ROLLBACK* message will cause a checkpointing handler to immediately stop all

activities related to checkpointing and join the recovery session. Since the first-level list is not cleared until the process has completed all phases of a checkpointing session, the recovery session is guaranteed to include all the handlers in the ongoing checkpointing session that may otherwise be waiting indefinitely for checkpointing-related responses. If a change in coordinator or session type is made then the handler propagates this decision to its first-level processes by resending *CHECKPOINT/ROLLBACK* messages with the new coordinator ID. Eventually the winning coordinator will ‘‘flush out’’ all remnants of the losing session.

However, a better protocol, discussed at length in Section 4.2.1.2, switches from checkpointing to recovery only when the process which has sent the *ROLLBACK* message has been positively determined to be part of the receiving process’ interacting set. This may involve postponing the response to the *ROLLBACK* message until more information is gathered (see Section 4.2.1.2). While this protocol is more complex, only the processes which *must* be rolled back will be rolled back.

4.4.4. Checkpointing with Packet-Switching

In a packet-switching environment messages are routed on a hop-by-hop basis over virtually any path in the system. Flushing of messages in transit during checkpointing is more complex since the entire system must be flushed. This requires temporarily stopping all the processes in the system from generating new messages while existing messages in transit are flushed to their final destinations and checks are performed on *every* communication link. As described by Tamir

and Gafni[Tami87], this is a classic distributed termination problem and the solution we use here is the same one used in [Tami87], which is derived from [Shav86, Dijk80].

4.5. Recovery

A recovery sessions is initiated when an error is detected. If there is a mismatch between the two signatures at the ends of a virtual circuit, a single interacting set needs to be rolled back: the set containing the two processes on either end of the virtual circuit. When an error in a node's outputs is detected by a neighboring node [Tami83], recovery may involve multiple interacting sets since all processes executing on the node and their interacting sets must be rolled back. Any messages lost due to the node's failure will be detected later when a signature mismatch occurs.

4.5.1. Recovery From Communication Errors

The algorithm to recover from a communication error is similar in structure to the *synchronous* checkpointing algorithm, where recovery coordinators and handlers replace their checkpointing counterparts. When a signature mismatch occurs, the kernel initiates rollback by spawning a *recovery coordinator* process. A *recovery tree* is created by propagating ROLLBACK messages which are acknowledged by RE_ACK(CHILD/NOT_CHILD) messages just as CHECKPOINT and CH_ACK messages are used. Unlike checkpointing, no signature comparisons are made, and all messages which are flushed to their destinations are discarded (this takes care of the recovery *livelock* problem

discussed in [Koo87]). For each process Y in the interacting set, the associated recovery handler RH_Y requests the process' state from the appropriate disk node. When a handler receives RE_DONE messages from all its children and its entire process state and associated message queues, it sends a RE_DONE message to its parent. When the coordinator receives its checkpointed state and RE_DONE messages from all its children, it sends RE_RESUME messages to it's children, marks its associated process ''runnable'', and terminates. Upon receipt of a RE_RESUME, RH_Y forwards the message to its children, terminates, and process Y resumes normal processing. No ''commit'' or ''resume acknowledgment'' phases are needed.

4.5.2. Recovery From Node Failures

When a neighbor detects that a node is faulty, *node-level recovery* is initiated. We assume that there is a reconfiguration algorithm that supplies the recovery algorithm with destination node(s) to which processes are to be restored. Since no information is available from the failed node(s), recovery trees must be constructed based on information available outside the failed node(s). An additional phase is added to the general structure of the algorithms to collect the missing information — namely, the list of first-level processes (*Buddies list*, see Procedure *Rback* in Section 4.1.5). In this phase, each neighbor which has detected the node failure starts up a Recovery Initiator Process (RIP_Y where Y is the *node* ID on which RIP_Y is running). The tasks performed by this process are: determining which processes were on the failed node(s), determining the first-level processes for each failed process, and starting Recovery Coordinators (phase two) for *each failed process*.

Phase two is identical to recovery from communication errors.

In order to determine which processes were on the failed node we require that the immediate neighbors of each node keep track (a list) of all processes running on the node. It should be noted that the disks must have lists of all processes in the system (the union of all the disk server process tables, see Figure 4.2 in the next subsection). Hence, if multiple nodes should fail simultaneously we can determine which processes are still in existence on the system and hence, which processes have failed. After determining which processes have failed, RIP_Y needs to get information equivalent to first-level process lists, for *each failed process*, before tree construction can begin. This is done by broadcasting RECOVERY messages (containing a list of failed processes) to all working nodes in the system. Each node returns a ‘‘FIRST-LEVEL message’’ which contains a list of processes on that node who have communicated directly with processes on the failed node (*Buddies list*). With this information all the required recovery trees can be constructed and recovery can proceed as in the previous subsection.

4.5.3. Handling Failures during Checkpoint Sessions

As discussed in Section 4.3.3, a recovery session has precedence over a checkpointing session. Specifically, if a recovery session has been chosen to supersede checkpointing, the checkpoint handlers stop all checkpointing activity and join the recovery session. For example, if the checkpointing coordinator receives a ROLLBACK message while waiting for CH_DONE messages from its children, it never sends a CH_COMMIT but instead immediately forwards the ROLLBACK message to all its children.

Each disk node has a disk server process that saves and restores checkpoints from the disk. The server process maintains a table that contains the status of the checkpoint of each process whose state is stored on the disk (Figure 4.2). Normally, a process has one entry, or checkpoint. During a checkpoint session, a second entry is made as the new checkpoint arrives at the disk node. This entry is invalid until the last state packet is received.

Disk Server Process Table

	PID	*Valid*	*New*	*Current CCid*	*Disk Address*	*Pointer*
A)	*F*	1	0	*B*		
B)	*F*	1	0	*B*		
		0	1	*A*		
C)	*F*	1	0	*B*		
		1	1	*A*		
D)	*F*	1	0	*A*		

Figure 4.2: The table maintained by the disk-server process on the disk node. A) shows an entry for process F between checkpoint sessions. Process B was F's previous checkpoint coordinator. The disk address field points to the location on disk of F's checkpoint. B) shows process F in the midst of a checkpoint session coordinated by process A. In C) F has two valid checkpoints and, in D), the newest state has been committed.

Each process has a *version* variable stored on the node where it is executing[Tami84]. When an error is detected this variable is used to determine

which version of the process checkpoint on disk should be used. The version variable has three possible values: *known, old,* and *unknown.* During normal operation, the version is always ‘‘known’’, meaning that there is only one valid checkpoint on disk. When the handler begins sending a new checkpoint state of a process to disk, the version variable changes to ‘‘old’’. When message queues are sent to disk, the handler changes the version to ‘‘unknown’’ and waits for a CH_COMMIT message. When the CH_COMMIT is received the handler sends a COMMIT message to the disk node. When the disk’s acknowledgment is received, the version is changed back to ‘‘known’’.

The version variable associated with the checkpoint *coordinator* never changes since the version is always ‘‘known’’, i.e., there is never more than one *valid* checkpoint state on disk. This means that the checkpoint coordinator never passes through step C in Figure 3, but moves directly from B to D — effectively committing the entire interacting set to the new checkpoint. This makes the checkpoint algorithm robust to failures. If any or even all processes in the interacting set fail, recovery coordinators look up the disk entry for the process being recovered — if there are two valid states for that process then the coordinator’s disk entry is used to determine which state to roll back to. If the coordinator’s entry has two states on disk, one valid and one not, then the process rolls back to its older checkpoint, while if there is only one valid entry the the process rolls back to its newer checkpoint.

4.6. Asynchronous Coordinated Checkpointing

Synchronous checkpointing of interacting sets requires that all processes (in the interacting set) be suspended for the duration of the checkpoint session. In the worst case the entire system may have to be suspended as in Global Checkpointing[Tami84]. The key feature of *asynchronous* checkpointing is that interruption of normal processing is minimized: the normal execution of a participating process is suspended only as long as it takes to copy its state in local memory (see Section 4.6 for further optimizations). Each process in the interacting set is ‘‘checkpointed’’ in *local volatile* memory and then resumes normal operation while the volatile checkpoints are copied to stable storage. Previous work has focused on optimizing the two-phase commit protocol in order to possibly reduce the length of time the process has to suspended[Kim93]. We eliminate the concern of overhead incurred by the two-phase commit structure of the checkpointing algorithm by making checkpointing activities asynchronous of process execution.

Checkpointing each process involves the following basic steps:

1. The process is halted. A *handler* process, that will participate in the checkpointing session on behalf of the halted process, is initiated.
2. The portion of the process state that is not in memory (e.g. contents of registers) is stored in a dedicated buffer area in local memory.
3. The process state in memory (or, as discussed in Section 4.6, just the modified pages) is copied to a dedicated buffer area for later transfer to stable storage.

4. *CHECKPOINT* messages are queued up for transmission to all neighbors (*Buddies*). (Any subsequent incoming messages will be delivered to the process *and*, possibly, saved as part of the checkpoint state.)

5. The process resumes normal operation.

6 The *handler* continues with the checkpointing session. Any outstanding packets in the communication channels to/from neighbors are flushed, the interacting set is identified, consistency of checkpoint and current states is maintained, and the coordinated checkpoint is committed to stable storage.

As in the synchronous checkpointing algorithm, checkpointing is triggered by a ‘‘checkpointing timer’’. The checkpoint handler assumes that it will coordinate the checkpoint session for the interacting set of processes to which it belongs. Processes that are not part of the interacting set do not participate in the checkpoint. The length of time that processes in the interacting set are suspended is *independent* of network delays *and* the number of processes that constitute the application. The cost is increased complexity in the initial phases of the checkpointing algorithm — until the interacting set has been found.

The checkpointing algorithm, with coordinator x, from the perspective of process p (p and x may be the same process), is as follows:

when either —

a) the checkpoint timer for process p expires ($x = p$) *or*

b) p has received its *first* $CHECKPOINT^x$ message from x or from $y \in Inter(x)$.

the following steps are taken:

i) p takes a checkpoint (C_p) (steps 2 and 3 above).

ii) The *Buddies*(p) list is formed: $q \in$ *Buddies*(p) if, and only if, process p has either sent a message to q or received a message from q since the last checkpoint of p.

iii) $CHECKPOINT^x$ messages are sent (queued) to all $q \in$ *Buddies*(p), except p's parent (if $p \neq x$). If $p \neq x$ then a CH_ACK^x message is sent (queued) to p's parent.

When any *marker* (*CHECKPOINT*, *CH_ACK*, or *SUBTREE–NC* message) is received by p from z, $MARKER_p[z]$=*TRUE*.

Until the interacting set has been found (step vii), if a *marker* has *NOT*:

a) been received from process z, then any application messages delivered to p (into $RecvQ_p[z]$, as discussed in Section 4.3.1) must also be saved in $MQ_p[z]$, or

b) been sent to process z, then p may queue messages to any z ($\notin$ *Buddies*(p)) but they must not be transmitted until a marker is sent or step vii is reached, whichever occurs first.

This step is critical to ensuring that the checkpoints of p and z will be consistent.

iv) ** p resumes normal execution.

v) p's handler now waits for CH_ACK^x and $SUBTREE^x$ messages from all $q \in$ *Buddies*(p), except for p's parent. *SUBTREE* messages are *SUBTREE-NC* (not-child) or *SUBTREE-C* (child).

vi) Once the *SUBTREE* messages have all arrived, $p \neq x$ sends a $SUBTREE-C^x$ message to its parent. If $p = x$ then the interacting set has been found and all in-transit messages have been flushed. Now, for each z from which a marker was received, $p = x$ saves $MQ_p[z]$ with C_p, and sends CH_FOUND^x messages to its children.

vii) Upon receipt of CH_FOUND^x from the parent, $MQ_p[z]$, for all z from which a marker has been received, must be saved with C_p. The CH_FOUND^x is forwarded to the children and p's handler waits for a) an acknowledgment from disk that the new checkpoint has been saved and b) CH_DONE^x messages from the children — indicating that *their* checkpoints have also been saved.

viii) Once the *CH_DONE* messages have all arrived *and* C_p is acknowledged to be stored on stable storage, p's handler sends a CH_DONE^x to its parent. If $p = x$ p directs its disk node to *commit* the most recent checkpoint and then sends CH_COMMIT^x messages to its children.

ix) Upon receipt of CH_COMMIT^x message from the parent, the handler directs the disk to commit the current checkpoint and CH_COMMIT^x is forwarded to the children. p's handler now waits for CH_RESUME^x messages from the children — indicating completion of their checkpoint handlers. x terminates once it receives the last CH_RESUME^x.

The *CHECKPOINT* and *CH_ACK* messages are used to locate the members of $Inter(x)$ and to ensure that the checkpoints of each pair of processes (p,q), where $p,q \in Inter(x)$, will be buddy-consistent. *SUBTREE* messages (child and not-child) are primarily used to complete the communication tree. *CH_FOUND*

messages, which travel down the tree, indicate that the tree is complete, the interacting set has been found, and that all messages have been flushed to their destinations (i.e. all the checkpoint states, *with MQ*, are buddy-consistent, and therefore form a consistent process set). Once stable storage has acknowledged receipt of each of the new checkpoints *CH_DONE* messages travel up the tree. The coordinator instructs the disk to commit to the new checkpoint (erasing the old) and then sends *CH_COMMIT* messages down the tree so that the rest of the interacting set will commit to their checkpoints. The checkpoint handlers terminate as *CH_RESUME* messages flow up the tree.

In synchronous coordinated checkpointing[Tami89, Koo87] processes do not resume normal operation and thus can not interfere with the *CHECKPOINT*, *CH_ACK*, and *SUBTREE* phases of the checkpoint session. In the proposed asynchronous coordinated checkpointing algorithm processes resume normal operation *prior* to these phases — increasing their complexity significantly. Important algorithmic details will be addressed in the following subsections: first we will look in more detail at the *CHECKPOINT*, *CH_ACK*, and *SUBTREE* phases of asynchronous checkpointing when 1) only one process (x) in the interacting set initiates (and coordinates) the checkpoint session and 2) only one checkpoint session is active in the system. In subsequent sections we will relax constraint 1) and then 2). At the end of the *CHECKPOINT*, *CH_ACK*, and *SUBTREE* phases the checkpoints of the interacting set form a consistent process set and the rest of the checkpoint session is simply the (robust) transfer of the checkpoint states to stable storage as discussed in Section 4.4.

4.6.1. Asynchronous Checkpointing: 1 Coordinator, 1 Session

In this section there is only one active checkpoint session with one coordinator, x. From the perspective of a process $p \in Inter(x)$ (where p and x may be the same process), every process q falls into one of the following three cases:

A) $q \in Buddies(p)$ and $q \in Inter(x)$:

Since q is a buddy of p, p has sent a marker, $CHECKPOINT^x$, to q. Step iii above and reliable transmission of recovery messages ensure that p will receive a marker from q.

When p receives either a 1) $CHECKPOINT^x$ or 2) CH_ACK^x then $MARKER_p[q]=TRUE$ and $MQ_p[q]$ is saved with C_p. In the first case p and q have ''simultaneously'' sent *CHECKPOINT* messages to one another. In order to create the interacting set tree, p declines child status with q by sending a $SUBTREE-NC^x$; q will do the same. In the second case q is a child of p thus p will receive a $SUBTREE-C^x$ message from q in the future.

B) $q \notin Buddies(p)$ and $q \in Inter(x)$:

This case is only interesting if q sends an application message to p before taking C_q (as part of $Inter(x)$) and that message is received by p after C_p is taken. Any such message(s) are saved in $MQ_p[q]$ (step iii-a above). When q takes C_q q sends $CHECKPOINT^x$ to p. Upon receipt of the $CHECKPOINT^x$ message p sets $MARKER_p[q]=TRUE$ and saves $MQ_p[q]$ with C_p. Note that since $q \notin Buddies(p)$, any messages sent by p to q have been blocked from leaving the node (step iii-b above). $SUBTREE-NC^x$ is sent to q and *then* any blocked messages to q are released.

C) $q \notin Buddies(p)$ and $q \notin Inter(x)$:

When p receives CH_FOUND^x from its parent any messages in $MQ_p[q]$ are discarded and any blocked messages to q are released.

This section solves the basic problems associated with ensuring that the checkpoint states of q and p are consistent. First, MQ is used to capture messages in transit ''between checkpoints'' so that, in the consistent global checkpoint state, all messages sent have been received (see example message $m1$ in Figure 4.3). Second, p has to be prevented from sending messages to any $q \notin Buddies(p)$ until either a) markers have been exchanged between p and q or b) p receives CH_FOUND^x from its parent. If q is allowed to receive messages from p where $q \notin Buddies(p)$ then p and q could end up with inconsistent checkpoint states as depicted by example message $m2$ in Figure 4.3. We show in Section 6.2 that very few messages needed to be blocked in our simulation runs.

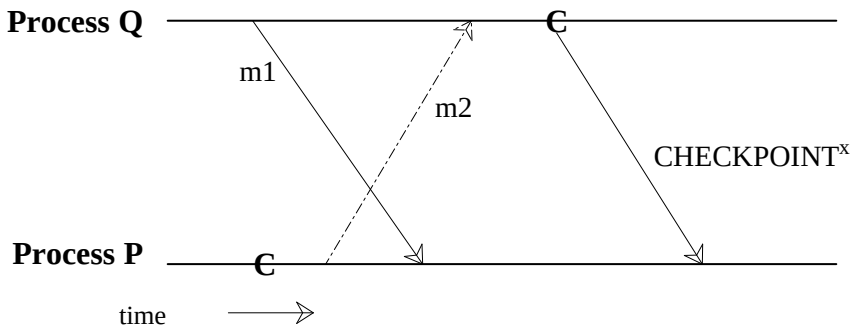

Figure 4.3: Inconsistent checkpoints. No communication between p and q occurred before p's checkpoint, thus q joined the checkpoint session as the buddy of some other process $r \in Inter(x)$. If q processes $m2$ before taking its checkpoint then q's checkpointed state has received a message which p's checkpoint never sent. This case is prevented from occurring by delaying transmission of $m2$ to q. Note that message $m1$ will correctly be saved in $MQ_p[q]$.

4.6.2. Asynchronous Checkpointing: N Coordinators, 1 Session

In this case multiple checkpoint handlers compete to coordinate the same checkpoint session. Again every process q falls into one of the following three cases:

A) $q \in Buddies(p)$ and $q \in Inter(x)$:

Besides 1) and 2) from the previous subsection, p could receive a 3) $CHECKPOINT^y$ from q. First, p sets $MARKER_p[q]$=$TRUE$ and saves $MQ_p[q]$ with C_p. If $y > x$ then q becomes p's new parent and p sends $CHECKPOINT^y$ to all $Buddies(p)$, except q, thereby propagating the change in coordinator. If $y < x$ then p does nothing. (Since $q \in Buddies(p)$, q will switch to coordinator x when it receives the $CHECKPOINT^x$ sent by p.)

B) $q \notin Buddies(p)$ and $q \in Inter(x)$:

In addition to 1) from the previous subsection, p might receive a 2) $CHECKPOINT^y$ from q. This marker can not be ''rejected'' because $q \in Inter(x)$ and therefore p and q must checkpoint together to ensure that the global checkpoint state remains consistent. Since, in this section, there is only one active checkpoint session p knows that q must be a member of $Inter(x)$. p responds by setting $MARKER_p[q]$=$TRUE$ and saving $MQ_p[q]$ with C_p. If $y > x$ then q becomes p's new parent and p then sends $CHECKPOINT^y$ to all $Buddies(p)$, except q. If $y < x$ p sends a $CHECKPOINT^x$ to q — forcing q to change coordinators.

C) $q \notin Buddies(p)$ and $q \notin Inter(x)$:

Same as in the previous subsection. Since we have only one checkpoint session active in the system (and q is not part of it since $q \notin Inter(x)$) then p can not

receive a $CHECKPOINT^{y}$ from q.

This section shows the basic mechanism for handling multiple processes attempting to coordinate *the same* checkpoint session. By imposing a total ordering amongst all process IDs we can guarantee that only one coordinator will ‘‘win’’ and that the checkpoint session will proceed.

Since asynchronous coordinated checkpointing allows processes to resume execution at the beginning of the checkpoint session it is essential that the very first *CHECKPOINT* message received from any process q, regardless of coordinator ID, result in *MARKER* being set to true. This first marker indicates that any previously-received messages were sent before q took its checkpoint and that any messages subsequently received were sent after q’s checkpoint. Once the marker has been received then any subsequent messages sent by q do not need to be saved in $MQ[q]$.

4.6.3. Asynchronous Checkpointing: N Coordinators, N Sessions

In this section we relax all constraints and allow multiple checkpoint handlers to compete to coordinate multiple checkpoint sessions. Every process q falls into one of the following three cases:

A) $q \in Buddies(p)$ and $q \in Inter(x)$: Same as in ‘‘N’’ Coordinators, 1 Session.

B) $q \notin Buddies(p)$ and $q \in Inter(x)$ and

C) $q \notin Buddies(p)$ and $q \notin Inter(x)$:

Now we can have the case that $q \in Inter(y)$ and $y \notin Inter(x)$. p can not tell the difference between B) and C) with its local $Buddies(y)$ information. After

receiving an application message from q p might receive either a 1) $CHECKPOINT^x$ or 2) $CHECKPOINT^y$ from q. If a $CHECKPOINT^x$ message is received then q is clearly in $Inter(x)$ and p responds as in 1 Coordinator, 1 Session, part B. If a $CHECKPOINT^y$ message is received p still does not know if $q \in Inter(x)$ or if $q \notin Inter(x)$. As discussed in the previous section p can not ''reject'' (negatively respond to) the $CHECKPOINT^y$. [Note: rejection is the approach taken in the synchronous algorithm presented in Koo and Toueg[Koo87] which must also deal with conflicting checkpoint sessions. We found, in our simulations, that this can result in many checkpoints never completing — creating too much ''useless'' overhead.]

Also, *unlike* the previous section p can not just positively respond to q's $CHECKPOINT^y$ either. The reason is as follows: Since $q \notin Buddies(p)$, p is not ''expecting'' a marker from process q. Therefore p can enter step vi of the algorithm and send a $SUBTREE-C$ to the parent. In effect p is telling its parent that ''the interacting set (tree) is complete from my perspective''. But, p might now receive a $CHECKPOINT^y$ from q. If p responds positively to this message then, if $q \notin Inter(x)$ p is *expanding* the interacting set to include q, thus ''violating'' its $SUBTREE-C$ message. With this approach p's coordinator has no way of knowing when the interacting set is complete. [Alternatively, a termination detection algorithm could be used to determine when the interacting set has been found. We decided this was too costly in terms of the time required to run the algorithm and the number of messages required.]

The last choice is for p to defer handling of $CHECKPOINT^y$. Several things can happen:

i) q changes to coordinator x and follows the $CHECKPOINT^y$ message with a $CHECKPOINT^x$. p then responds as in 1 coordinator, 1 session, part A.

ii) p changes to coordinator y and then responds to q as in 1 coordinator, 1 session, part A.

In i) and ii) q is clearly a member of $Inter(x)$.

iii) If $q \notin Inter(x)$ then p receives a CH_FOUND^x from its parent. At this time p finally knows that $q \notin Inter(x)$: p completes its current checkpoint and afterwards joins q's checkpoint as q's child.

iv) If $q \in Inter(x)$ deadlock can occur, as shown in Figure 4.4. To prevent deadlock we add the following to our algorithm. When p, checkpointing with coordinator x, decides to defer handling of a $CHECKPOINT^y$ message, p sends two messages — one to each coordinator — relaying the fact that p is not responding to a $CHECKPOINT^y$ message. If, for example (see Figure 4.4), coordinator x receives *two* messages: one from a process $r \in Inter(y)$ indicating that r can not respond to a $CHECKPOINT^x$ message and one from a process $p \in Inter(x)$ indicating that p can not respond to a $CHECKPOINT^y$ message, then x knows that $Inter(x)=Inter(y)$. If $y>x$ then x changes coordinators, making y its parent and sending $CHECKPOINT^y$ messages to all $Buddies(x)$. If $y<x$ then x does nothing — knowing that y will switch coordinators to x and become x's child.

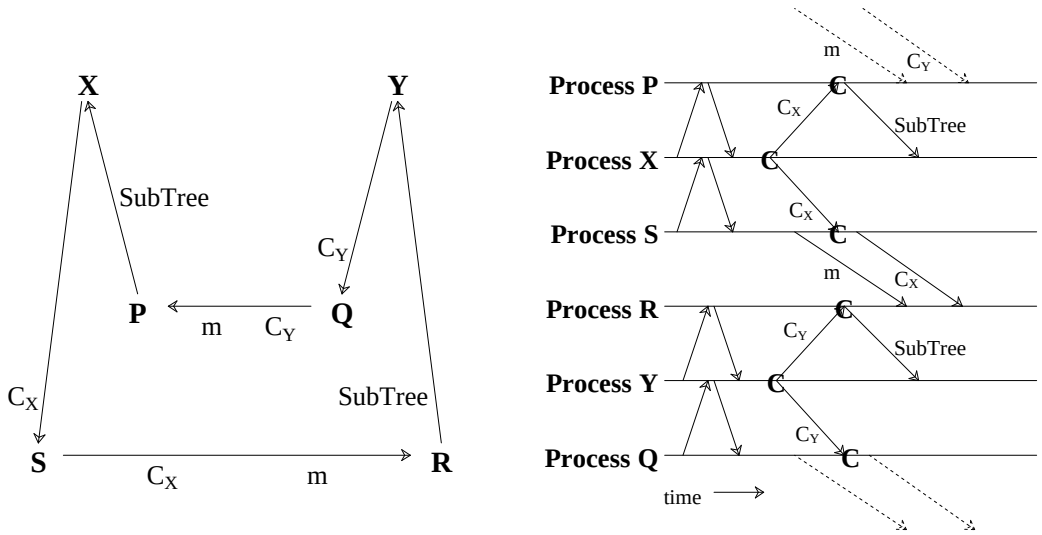

Figure 4.4: Deadlocked checkpoint session. The cycle is depicted on the left and the details of the example on the right. The problem is that process Q can not send a $SUBTREE-C$ message to Y without receiving a $SUBTREE$ from P. The same situation exists for S. P (and R) can not respond because they have already informed their respective coordinators that their portion of the interacting set (tree) is complete. Thus the algorithm is deadlocked and can not proceed.

4.6.4. Summary

We have now finished detailing the checkpointing algorithm for the general case. At the end of the *CHECKPOINT*, *CH_ACK*, and *SUBTREE* phases the checkpoints of the interacting set form a consistent process set. The robust transfer of these states to stable storage was addressed in Section 4.4. In coordinated checkpointing algorithms processes not part of the interacting set do not participate in the checkpoint sessions. In the asynchronous algorithm the length of time that a process is suspended is independent of network delays and the number of processes in the interacting set. This is similar to algorithms that checkpoint processes independently except that processes in our algorithm will, for a given checkpoint

interval, checkpoint more often as they participate in checkpoint sessions that they did not initiate. This can be tuned by increasing the checkpoint interval so that, on average the *actual* checkpoint interval is the desired checkpoint interval. Section 6.2 contains a performance comparison of independent and coordinated checkpointing, in a failure-free system.

The cost of asynchronous coordinated checkpointing compared to synchronous is the need to (possibly) delay some messages from being transmitted and an increase in the complexity of the *CHECKPOINT*, *CH_ACK*, and *SUBTREE* phases of the checkpoint algorithm. We have proven the correctness of synchronous coordinated checkpointing and have focused our efforts on implementing the asynchronous algorithm on a parallel simulator.

4.7. Minimizing the Cost of Checkpointing Using the Virtual Memory System

Asynchronous coordinated checkpointing is amenable to several optimizations that can dramatically decrease the amount of data that must be moved during checkpointing as well as the local memory space needed for the volatile checkpoints. The key to these optimization is to use an enhanced virtual memory system. This system will help the scheme in two ways: 1) it will identify the pages that have been modified since the last checkpoint and thus eliminate the need to checkpoint pages that have not changed, and 2) it will allow the volatile checkpoint to be taken by simply copying page table entries without actually moving data in local memory unless it becomes necessary.

The first optimization requires adding a special ‘‘dirty bit’’ to each entry in the page table. This dirty bit is cleared when a process is checkpointed and is set

whenever there is a *write* to the page. When a page is first allocated, is is marked as ‘‘dirty.’’ During checkpointing, only dirty pages that are in local memory need to be physically copied to stable storage for checkpointing. Pages that are ‘‘clean’’ according to this special dirty bit have not changed since the last checkpoint and are included in the new checkpoint in stable storage using simple pointer manipulation (there is no need to move the page from the physical location in stable storage where it was stored for the previous checkpoint).

In standard virtual memory systems, page table entries include control bits that mark the page as *read-only*, *read-write*, *execute-only*, etc. The time to take the volatile checkpoint as well as the space required for it in local memory can be cut dramatically by adding another type of page: *copy-on-write* [Bobr72]. Taking a volatile checkpoint involves copying the page table entries of resident dirty pages and marking the corresponding entries in the page table of the process being checkpointed as *copy-on-write*. As long as the process does not actually attempt to write into the page, the page will *not* be copied to the volatile checkpoint area in local memory. Instead, as long as the process is only reading the page, the checkpointing handler, that is copying the page to stable storage, and the process using the page for normal processing will share the page. If the process tries to write to the page, a special page fault will be triggered and the page will be copied to a different location in memory. On the other hand, if the checkpointing session completes before the process attempts to write to the page, the page table entry is restored to its previous state and any writes by the process will *not* trigger a page fault. It should be noted that this very promising optimization can *only* be used with asynchronous checkpointing. Previous checkpointing techniques block

process execution until checkpointing is complete so that ‘‘sharing’’ of pages between the process and the checkpointing handler is meaningless.

In a typical system pages containing code are usually read-only, loads are more frequent than stores, and a checkpointing session is expected to take significantly less time than the interval between checkpoints. Based on these facts we expect the optimizations described above to significantly reduce both the time to take a volatile checkpoint and the time to commit the checkpoint to stable storage.

4.8. Conclusions

We have presented two new distributed error recovery schemes for multicomputers based on coordinated checkpointing of interacting sets that minimize disruptions to normal processing due to checkpointing. The domino effect can not occur and only one generation of checkpoints needs to be maintained on stable storage. The algorithms have been designed to handle both communication errors and node failures.

In the synchronous checkpointing scheme an interacting set of processes is suspended from normal computation until all of the process states have been committed to stable storage. The asynchronous scheme is based on (coordinated) volatile checkpointing of interacting sets of processes followed by copying of the checkpoints to stable storage ‘‘in the background’’ involving minimal interference with normal computation. The schemes are integrated with an efficient error detection mechanism that avoids the need for message acknowledgments, transmission of message sequence numbers, and transmission of check bits with

each message. Both schemes can recover from multiple node failures, lost messages, and corrupt messages. Multiple checkpointing and recovery sessions may be active in the system simultaneously operating independently if possible and merging correctly when necessary.

Additionally, as discussed in Section 4.6, the asynchronous algorithm is amenable to highly-effective optimizations that can be supported with relatively small enhancements to a conventional virtual memory system. With these optimizations, a volatile checkpoint can be taken by only a few dozen instructions without physically copying any pages in local memory.

Using the proposed techniques it is possible to implement a highly reliable, general-purpose, large multicomputer system in which the fault tolerance characteristics involve minimal performance overhead and are completely transparent to the user.