Chapter Two
System Model

The target system for which error recovery is required is a multicomputer consisting of hundreds or thousands of VLSI nodes which communicate via messages over point-to-point links. In the first section of this chapter we describe the computational model of the target system. This includes several basic ideas and key definitions which are used throughout the thesis and which serve as a foundation for proving the correctness of the presented recovery algorithms. In Section 2.2 the characteristics of the multicomputer are described and some assumptions are made. Section 2.3 describes the interprocess communication model used by the multicomputer. The final section details the error detection techniques that are used to detect node failures and communication failures; all message logging algorithms assume that communication is reliable while the presented coordinated checkpointing algorithms use a less restrictive model where the detection of communication failures can be incorporated into the error recovery algorithms — resulting in a potentially significant reduction in overhead.

2.1. Computational Model

In this thesis we use a combination of the distributed system models presented in [Chan85] and [Stro85].

2.1.1. Fault-Free Operation

As in [Chan85] a distributed system consists of a finite set of n processes and a finite set of channels. A process is defined by a set of states, including an initial state, and a set of events. The state of a process consists of all of the storage elements associated with than process. Processes communicate via directed channels which deliver error-free messages in FIFO order and with arbitrary finite delay. The state of a channel is determined by the sequence of messages sent along it excluding the messages received along it. An *event* is the cause of a state change and is either: 1) an internal process event, 2) the receipt of a message, or 3) the sending of a message. If a send event for a message m has occurred in some process then a corresponding receive event must eventually occur in some other process.

A *global state* of a distributed system is a set of n component process and channel states [Chan85]. The *initial global state* is one in which the state of each process is its initial state and the state of each channel is empty.

The *system history* is the set of executed events. The *depends-on*[Stro85] relation (*happens before*[Lamp78]) describes the relationship between two events a and b. An event b depends on event a $(a \rightarrow b)$ if and only if a and b are events in the same process and a occurs before b, or a is the sending of a message m by a process and b is the receipt of that message by another process. If b depends on a and c depends on b, then c depends on a (transitive closure), while if a does not depend on b and b does not depend on a, then the two events are said to be *concurrent*. A *valid* system history is a system history for which the depends-on relation is an irreflexive partial ordering[Schn85].

2.1.2. Operation with Failures and Recovery

When a failure occurs the system must be restored to a global state that is a possible intermediate state in fault-free execution. Such an intermediate state (a *consistent global state*[Chan85]) is composed entirely of *consistent* process and channel states such that every pair of processes and the connecting channels ‘‘agree’’ on the number and content of messages which have been exchanged (see Figure 2.1). States can become inconsistent in two ways:

1) If a message is *lost* (sent but not received) correctness is violated since, if a message is sent it must eventually be received. Correctness is restored by reconstructing and (re)delivering the message or by undoing the send event. and,

2) If a message is *orphaned* (received but not sent) correctness is violated because the system history is no longer valid. Validity is restored by reconstructing the send event or undoing the receive event.

Checkpointing and rollback of process and channel states can be used to provide transparent error recovery for application processes. Recording a process state is not considered to be an event, therefore checkpointing does not alter the ordering of events in the system history. When a failure occurs the global system state may contain inconsistent process and channel states due to lost or orphaned messages.

Message logging techniques use checkpoints, message logs, and ordering information (the depends-on relation) to roll back failed process(es) and then *adjust* the state of that process to be consistent with the rest of the system state. Some non-failed process states may be *orphaned* (see Figure 2.1) and must also be rolled

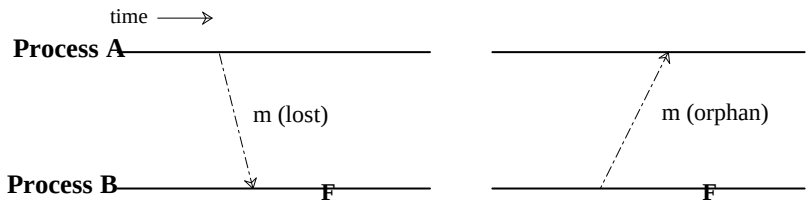

Figure 2.1: In both cases process B has failed and its state has been lost. In the first case A has sent a message for which there is no corresponding receive event (i.e. m has been *lost*). A consistent global state requires that the message not be lost (e.g. m was logged and will be received) or else that the sending of the message be undone (e.g. roll A back). In the second case A has received a message for which there has been no corresponding send event. A is an *orphaned computation*[Lisk87] (or *orphan*) that must be undone unless the send event is reconstructed.

back because inconsistent information (orphan message(s)) has been incorporated into their states.

Message logging techniques require applications to be *deterministic* in their actions: given a process state (checkpoint) and a sequence of inputs (message log), the process will generate the same outputs. When a failure occurs, the failed process(es) is rolled back and its message log is replayed. Ideally, all the messages that the process had received plus ordering information are retrievable from stable storage, in which case their subsequent replay results in the process recovering completely and identically to the point at which it previously failed. As long as duplicate outputs are detected and deleted, no other processes are affected by the rollback and subsequent replay. Synchronous message logging techniques achieve this by logging all input messages synchronously to stable storage and taking a potentially significant performance penalty during normal operation. However, if input messages are logged asynchronous of process execution, then recovery must

result in a consistent system state despite possible lost messages (not logged at the time of failure) and orphaned process states.

Coordinated checkpointing techniques maintain a *consistent global state* on stable storage. Then, 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.

With the *global* coordinated checkpointing technique, a checkpoint session begins by flushing all messages in transit to their destinations, after which the entire system state is saved to stable storage. A recovery session simply flushes and discards any messages in transit and retrieves the saved global state.

Global checkpointing has been extended to perform checkpointing and recovery on smaller portions of the system — sets of interacting processes.

Definition 2.1:

An *interacting set of processes* is the set of processes which have communicated directly or indirectly since their last checkpoint such that,

If p is a member:

Any process q that has communicated with p since q's last checkpoint is also a member.

It follows from this definition that an interacting set of processes forms a strongly connected system graph where the processes in the interacting set are vertices and the communication(s) between them are arcs.

Thus, a distributed system is composed of a collection of *disjoint* interacting sets of processes. Since there has been no communication between processes in

different sets since their last checkpoint, a new checkpoint of a process in one interacting set is consistent with both the old and new checkpoint of another process which is in some other interacting set. Hence, different interacting sets may be checkpointed and recovered independently and a consistent global state (checkpoint) is always maintained.

These basic ideas are extended in the chapters on message logging and coordinated checkpointing techniques (Chapters 3 and 4, respectively) where we use this model to prove the correctness of our recovery algorithms.

2.2. Assumptions

The target system requiring fault tolerance is a multicomputer consisting of hundreds of nodes which communicate via messages over point-to-point links. Each node includes a processor, local memory, and a communication coprocessor. The nodes operate asynchronously and messages may have to pass through several intermediate nodes on their way to their destinations. The system is used for ‘‘general purpose’’ applications which have no hard real-time constraints. Errors can occur at any time as a result of *hardware* faults in the nodes or in the communication links.

The error recovery scheme is based on the existence of stable storage where checkpoints can be safely maintained. Such stable storage may be implemented as mirrored disks. Some of the nodes in the system, which we call *disk nodes*, are connected to such ‘‘reliable’’ disks. We assume that a failure of the disks themselves or of the disk nodes causes a *crash* (i.e., an unrecoverable error).

The connectivity of the system is high so that the probability of the system

partitioning due to the failure of a node(s) is low enough so that it is reasonable for partitioning to cause a *crash*.

The *state* of a process, which gets checkpointed periodically and recovered once an error is detected, is the contents of all of the memory and registers used by the process. This includes some system tables, such as the list of all the virtual circuits currently established to and from the process.

Since each node can be time-shared between multiple processes, it may have to participate in multiple simultaneous checkpointing and recovery sessions. Hence, it is not advisable to implement checkpointing and recovery as part of the kernel. Instead, whenever checkpointing or recovery of a particular process is initiated, the kernel spawns a special *handler* process that performs the necessary operations. The handler can suspend the process, manipulate its state, and allow it to resume normal operation. In the rest of the thesis we will often discuss the actions of participants in checkpointing and recovery sessions. These ‘‘participants’’ are really the handlers corresponding to the processes being checkpointed or recovered.

2.3. Interprocess Communication

A fault-tolerant multicomputer must be able to detect and recover from errors in interprocessor communication. The message delivery mechanism has a strong effect on the performance of checkpointing and recovery schemes. Many error recovery schemes [Stro85, Koo87] are based on the use of reliable FIFO communication channels between nodes. These channels are implemented by underlying communication protocols which typically involve sending redundant

bits for error detection, appending sequence numbers to each message, and the use of end-to-end acknowledgments [Tane81]. The message logging algorithm described in this thesis (Chapter 3) is based on this kind of reliable protocol.

For the coordinated checkpointing algorithms presented in this thesis (Chapter 4) two communication mechanisms (containing no facilities for error detection) are considered: virtual circuits and simple packet switching. In a system where processes communicate using *virtual channels* (or virtual circuits) [Bert87, Reed87], i.e., if processes on two nodes that are not immediate neighbors need to communicate, a logical circuit is set up from the source to the destination by placing appropriate entries in the routing tables of each intermediate node along the way. Once the path is set up, there is very little routing overhead for packets sent through the circuit and FIFO ordering of these messages is maintained. With message/packet switching no path is established in advance between the sender and the receiver. Every packet is routed independently at each hop and packets (messages) may arrive at their destination out of order. We assume that the operating system will reorder packets as necessary.

2.4. Error Detection

As previously mentioned, errors in the system may be a result of node failures or failures in the communication links. We assume that the nodes are self-checking and produce an error indication whenever their outputs are incorrect[Tami83].

In most systems, errors in message transmission are detected by including with each message *check bits*, which the receiver uses to determine whether the

contents of a message has been corrupted. Lost messages are detected by protocols that involve acknowledging each message as well as transmission of sequence numbers with each message[Tane81]. The disadvantage of these techniques is that they involve transmission of redundant bits and thus ‘‘waste’’ communication bandwidth. Since the probability of an error in transmission is low, it is wasteful to check the validity of each message or packet independently. For the coordinated checkpointing algorithms the following technique is used.

As proposed in[Tami84], each node has two special purpose registers for error detection associated with each of its ports. One of these registers contains the CRC (Cyclic Redundancy Check) check bits for all the packets that have been sent from the port. The other register contains the CRC check bits for all packets received. These special purpose registers are linear feedback shift registers (LFSRs) and their contents are updated in parallel with the transmission of each packet[Elki82].

In order to check the validity of all the packets transmitted through a particular link, each node sends to its neighbor the contents of the LFSR used for outgoing packets. The neighbor can then compare the value it receives with the value in its LFSR for *incoming* packets and signal an error if it finds a mismatch. In this scheme, if packet switching is used, *all* the links in the system must be checked in this way before committing to a new checkpoint. Otherwise, the state of a node corrupted by an erroneous message may be checkpointed and later used for recovery.

Alternatively, *virtual LFSRs* can be maintained. With virtual circuits, a virtual LFSR is used to accumulate the signature of the packets transmitted from the source process of a virtual circuit and another virtual LFSR is used to

accumulate the signature of packets received by the destination process of that virtual circuit. Communication between processes in the interacting set can then be checked without checking all the links in the system, by performing ‘‘end-to-end’’ checks on all the virtual circuits between processes in that set. Each node might, for example, maintain a cache of virtual LFSRs by those circuits which have either a source or destination process at that node. Hence, the LFSRs are not updated by packets being transmitted *through* a node but only for those whose source or destination is the current node. If packet switching is used the same technique can be employed despite the fact that messages may take ‘‘any’’ path to get to their destination.

The packets used to coordinate the creation of checkpoints and for error recovery must be verified before they are used. Hence, for these packets, an error detecting code is used and redundant bits are transmitted with the packet. Thus, there are two types of packets in the system: normal packets that do not include any information for error detection, and special control packets, called *fail-safe* packets, that are used only for transmitting information between handlers and which include a sufficient number of redundant bits to detect likely errors in transmission. The *fail-safe* packets are either error-free or the error is easily detectable by the receiving node.

When a node fails, recovery involves rolling back the union of the interacting sets of all the processes that were running on that node and requires eventual roll back of all interacting sets of processes which had any messages in transit on that node at the time of its failure. When an error is detected by a mismatch of the LFSRs on two ends of a physical link and *packet switching* is used, the entire

system must be rolled back since there is no way to determine which processes were affected by corrupt messages. If virtual circuits are used, and an error is detected by a mismatch of signatures on the two ends of a virtual circuit then only the interacting set which includes the two processes connected by that virtual circuit need to be rolled back. See Chapter 4, Section 4.4 for more details.