- Recovery

Fundamental to fault tolerance is the recovery from an error. Recall that an error is that part of a system that may lead to a failure. The whole idea of error recovery is to replace an erroneous state with an error-free state. There are essentially two forms of error recovery.

In backward recovery, the main issue is to bring the system from its present erroneous state back into a previously correct state. To do so, it will be necessary to record the system's state from time to time, and to restore such a recorded state when things go wrong. Each time (part of) the system's present state is recorded, a checkpoint is said to be made.

Another form of error recovery is forward recovery. In this case, when the system has entered an erroneous state, instead of moving back to a previous, checkpointed state, an attempt is made to bring the system in a correct new state from which it can continue to execute. The main problem with forward error recovery mechanisms is that it has to be known in advance which errors may occur.

Only in that case is it possible to correct those errors and move to a new state. The distinction between backward and forward error recovery is easily explained when considering the implementation of reliable communication. The common approach to recover from a lost packet is to let the sender retransmit that packet. In effect, packet retransmission establishes that we attempt to go back to a previous, correct state, namely the one in which the packet that was lost is being sent. Reliable communication through packet retransmission is therefore an example of applying backward error recovery techniques.

An alternative approach is to use a method known as erasure correction. In this approach. a missing packet is constructed from other, successfully delivered packets. For example, in an (n,k) block erasure code, a set of k source packets is encoded into a set of n encoded packets, such that any set of k encoded packets is enough to reconstruct the original k source packets. Typical values are k =16' or k=32, and k<11~2k [see, for example, Rizzo (1997)]. If not enough packets have yet been delivered, the sender will have to continue transmitting packets until a previously lost packet can be constructed. Erasure correction is a typical example of a forward error recovery approach.

By and large, backward error recovery techniques are widely applied as a general mechanism for recovering from failures in distributed systems. The major benefit of backward error recovery is that it is a generally applicable method independent of any specific system or process. In other

words, it can be integrated into (the middleware layer) of a distributed system as a general-purpose service.

However, backward error recovery also introduces some problems (Singhal and Shivaratri, 1994). First, restoring a system or process to a previous state is generally a relatively costly operation in terms of performance. As will be discussed in succeeding sections, much work generally needs to be done to recover from, for example, a process crash or site failure. A potential way out of this problem is to devise very cheap mechanisms by which components are simply rebooted. We will return to this approach below.

Second, because backward error recovery mechanisms are independent of the distributed application for which they are actually used, no guarantees can be given that once recovery has taken place, the same or similar failure will not happen again. If such guarantees are needed, handling errors often requires that the application gets into the loop of recovery. In other words, full-fledged failure transparency can generally not be provided by backward error recovery mechanisms.

Finally, although backward error recovery requires checkpointing, some states can simply never be rolled back to. For example, once a (possibly malicious) person has taken the $1.000 that suddenly came rolling out of the incorrectly functioning automated teller machine, there is only a small chance that money will be stuffed back in the machine. Likewise, recovering to a previous state in most UNIX systems after having enthusiastically typed rrn -fr * but from the wrong working directory, may turn a few people pale. Some things are simply irreversible.

Checkpointing allows the recovery to a previous correct state. However, taking a checkpoint is often a costly operation and may have a severe performance penalty. As a consequence, many fault-tolerant distributed systems combine checkpointing with message logging. In this case, after a checkpoint has been taken, a process logs its messages before sending them off (called sender-based logging). An alternative solution is to let the receiving process first log an incoming message before delivering it to the application it is executing. This scheme is also referred to as receiver-based logging. When a receiving process crashes, it is necessary to restore the most recently checked state, and from there on replay the messages that have been sent. Consequently, combining checkpoints with message logging makes it possible to restore a state that lies beyond the most recent checkpoint without the cost of checkpointing.

Another important distinction between checkpointing and schemes that additionally use logs follows. In a system where only checkpointing is used, processes will be restored to a checkpointed state. From there on, their behavior may be different than it was before the failure

occurred. For example, because communication times are not deterministic, messages may now be delivered in a different order, in turn leading to different reactions by the receivers. However, if message logging takes place, an actual replay of the events that happened since the last checkpoint takes place. Such a replay makes it easier to interact with the outside world, For example, consider the case that a failure occurred because a user provided erroneous input. If only checkpointing is used, the system would have to take a checkpoint before accepting the user's input in order to recover to exactly the same state. With message logging, an older checkpoint can be used, after which a replay of events can take place up to the point that the user should provide input.

In practice, the combination of having fewer checkpoints and message logging is more efficient than having to take many checkpoints.

Stable Storage

To be able to recover to a previous state, it is necessary that information needed to enable recovery is safely stored. Safely in this context means that recovery information survives process crashes and site failures, but possibly also various storage media failures. Stable storage plays an important role when it comes to recovery in distributed systems. We discuss it briefly here.

Storage comes in three categories. First there is ordinary RAM memory, which is wiped out when the power fails or a machine crashes. Next there is disk storage, which survives CPU failures but which can be lost in disk head crashes.

Finally, there is also stable storage, which is designed to survive anything except major calamities such as floods and earthquakes. Stable storage can be implemented with a pair of ordinary disks, as shown in Fig. 8-23(a). Each block on drive 2 is an exact copy of the corresponding block on drive 1. When a block is updated, first the block on drive 1 is updated and verified. then the same block on drive 2 is done.

Suppose that the system crashes after drive 1 is updated but before the update on drive 2, as shown in Fig. 8-23(b). Upon recovery, the disk can be compared block for block.

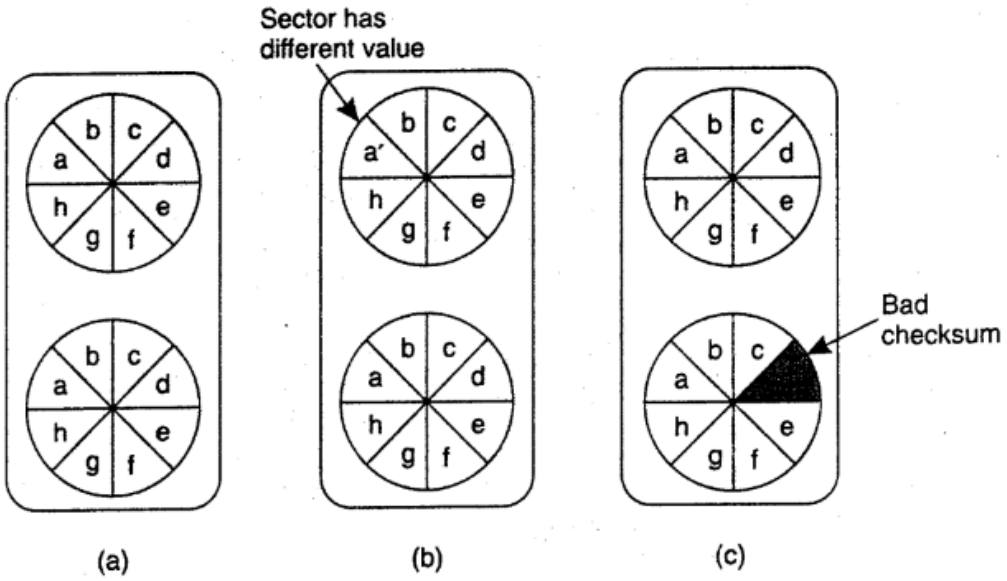

Figure 8-23. (a) Stable storage. (b) Crash after drive 1 is updated. (c) Bad spot.

Whenever two corresponding blocks differ, it can be assumed that drive 1 is the correct one (because drive 1 is always updated before drive 2), so the new block is copied from drive 1 to drive 2. When the recovery process is complete, both drives will again be identical.

Another potential problem is the spontaneous decay of a block. Dust particles or general wear and tear can give a previously valid block a sudden checksum error, without cause or warning, as shown in Fig. 8-23(c). When such an error is detected, the bad block can be regenerated from the corresponding block on the other drive.

As a consequence of its implementation, stable storage is well suited to applications that require a high degree of fault tolerance, such as atomic transactions.

When data is written to stable storage and then read back to check that it has been written correctly, the chance of it subsequently being lost is extremely small.