FAULT TOLERANCE

A characteristic feature of distributed systems that distinguishes them from single-machine systems is the notion of partial failure.

A partial failure may happen when one component in a distributed system fails. This failure may affect the

proper operation of other components, while at the same time leaving yet other components totally unaffected.

In contrast, a failure in nondistributed systems is often total in the sense that it affects all components, and may easily bring down the entire system.

An important goal in distributed systems design is to construct the system in such a way that it can automatically recover from partial failures without seriously affecting the overall performance.

In particular, whenever a failure occurs, the distributed system should continue to operate in an acceptable way while repairs are being made, that is, it should tolerate faults and continue to operate to some extent even in their presence.

Basic Concepts

To understand the role of fault tolerance in distributed systems we first need to take a closer look at what it actually means for a distributed system to tolerate faults.

1. Availability
2. Reliability
3. Safety
4. Maintainability

Avail ability is defined as the property that a system is ready to be used immediately.

Reliability refers to the property that a system can run continuously without failure.

Safety refers to the situation that when a system temporarily fails to operate correctly, nothing catastrophic happens.

Finally, maintainability refers to how easy a failed system can be repaired. A -highly maintainable system may also show a high degree of availability, especially if failures can be detected and repaired automatically

Failure Models

A system that fails is not adequately providing the services it was designed for. If we consider a distributed system as a collection of servers that communicate with one another and with their clients, not adequately providing services means that servers, communication channels, or possibly both, are not doing what they are supposed to do.

However, a malfunctioning server itself may not always be the fault we are looking for. If such a server depends on other servers to adequately provide its services, the cause of an error may need to be searched for somewhere else.

Type of failure	Description
Crash failure	A server halts, but is working correctly until it halts
Omission failure *Receive omission* *Send omission*	A server fails to respond to incoming requests A server fails to receive incoming messages A server fails to send messages
Timing failure	A server's response lies outside the specified time interval
Response failure *Value failure* *State transition failure*	A server's response is incorrect The value of the response is wrong The server deviates from the correct flow of control
Arbitrary failure	A server may produce arbitrary responses at arbitrary times

Figure 8-1. Different types of failures.

Failure Masking Redundancy

If a system is to be fault tolerant, the best it can do is to try to hide the occurrence of failures from other processes. The key technique for masking faults is to use redundancy.

Three kinds are possible: information redundancy, time redundancy, and physical redundancy [see also Johnson (1995)].

With information redundancy, extra bits are added to allow recovery from garbled bits. For example, a Hamming code can be added to transmitted data to recover from noise on the transmission line.

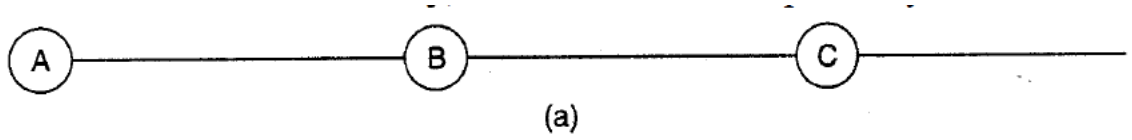
With time redundancy, an action is performed, and then. if need be, it is performed again. Transactions (see Chap. 1) use this approach.

If a transaction aborts, it can be redone with no harm.

Time redundancy is especially helpful when the faults are transient or intermittent. With physical redundancy, extra equipment or processes are added to make it possible for the system as a whole to tolerate the loss or malfunctioning of some components.

Physical redundancy can thus be done either in hardware or in software. For example, extra processes can be added to the system so that if a small number of them crash, the system can still function correctly

Physical redundancy is a well-known technique for providing fault tolerance.
It is used in biology (mammals have two eyes, two ears, two lungs, etc.), aircraft (747s have four engines but can fly on three), and sports (multiple referees in case one misses an event).
It has also been used for fault tolerance in electronic circuits for years; it is illustrative to see how it has been applied there.
Consider, for example, the circuit of Fig. 8-2(a).
Here signals pass through devices *A, B,* and C, in sequence. If one of them is faulty, the final result will probably be incorrect.



(a)

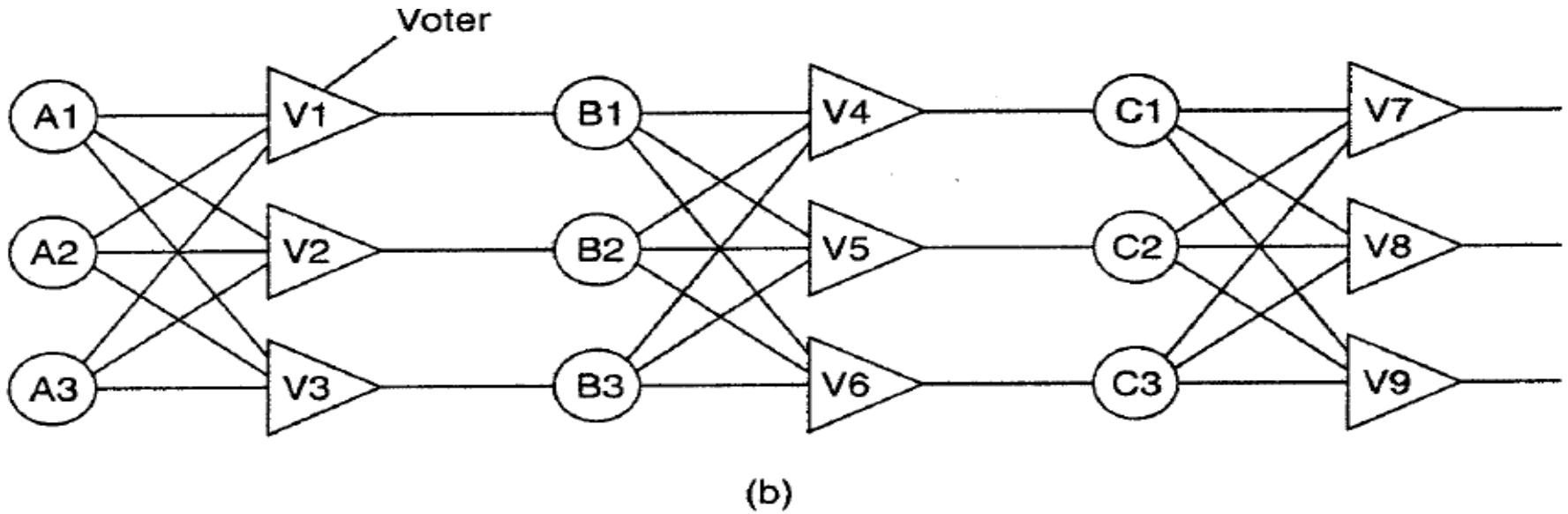


Figure 8-2. Triple modular redundancy.

In Fig. 8-2(b), each device is replicated three times. Following each stage in the circuit is a triplicated voter. Each voter is a circuit that has three inputs and one output. If two or three of the inputs are the same, the output is equal to that input. If all three inputs are different, the output is undefined. This kind of design is known as **TMR** (Triple **Modular** Redundancy).

Suppose that element A_Z fails. Each of the voters, Vb V_Z, and V_3 gets two good (identical) inputs and one rogue input, and each of them outputs the correct value to the second stage. In essence, the effect of A_Z failing is completely masked, so that the inputs to B_1, B_Z, and B_3 are exactly the same as they would have been had no fault occurred.

Now consider what happens if B_3 and C_1 are also faulty, in addition to A_2. These effects are also masked, so the three final outputs are still correct.

At first it may not be obvious why three voters are needed at each stage. After all, one voter could also detect and pass though the majority view. However, a voter is also a component and can also be faulty. Suppose, for example, that voter V_1 malfunctions. The input to B_1 will then be wrong, but as long as everything else works, B_2 and B_3 will produce the same output and V_4, V_5, and V_6 will all produce the correct result into stage three. A fault in V_1 is effectively no different than a fault in B_1. In both cases B_1 produces incorrect output, but in both cases it is voted down later and the final result is still correct.

PROCESS RESILIENCE

Flat Groups versus Hierarchical Groups

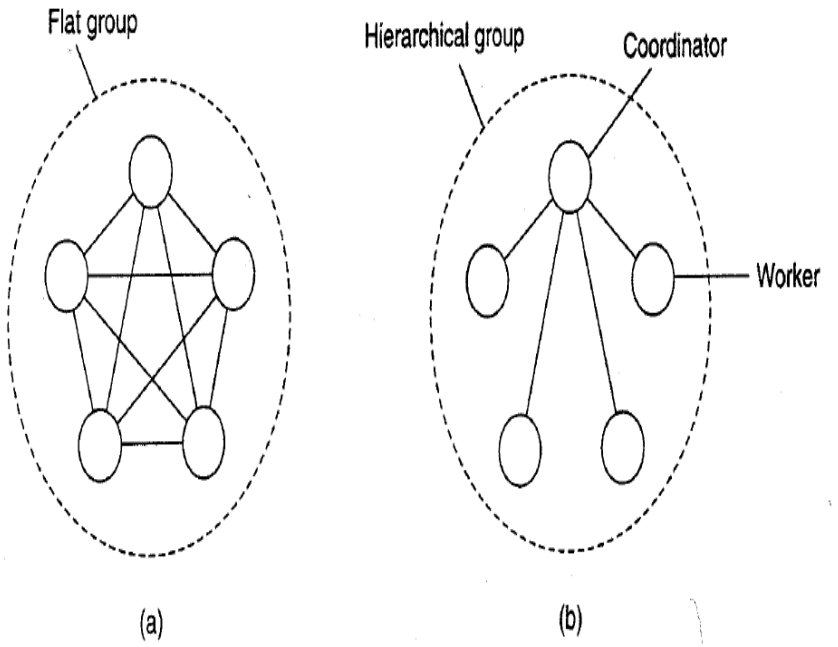


Figure 8-3. (a) Communication in a flat group. (b) Communication in a simple hierarchical group.

Each of these organizations has its own advantages and disadvantages. The flat group is symmetrical and has no single point of failure. If one of the processes crashes, the group simply becomes smaller, but can otherwise continue.

A disadvantage is that decision making is more complicated. For example, to decide anything, a vote often has to be taken, incurring some delay and overhead.

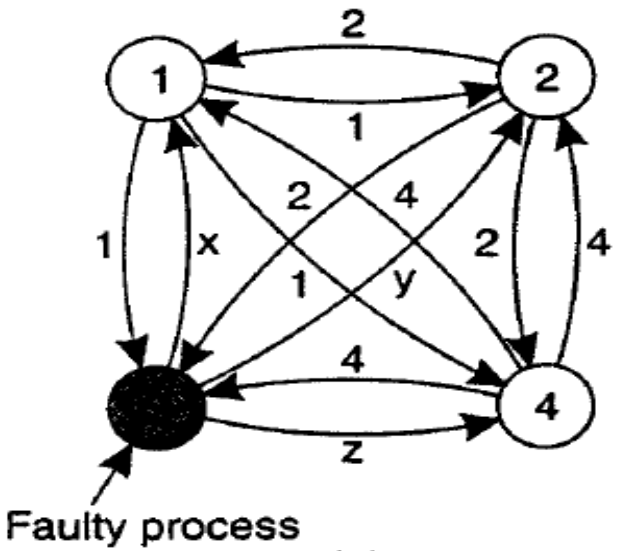
The hierarchical group has the opposite properties. Loss of the coordinator brings the entire group to a grinding halt, but as"long as it is running, it can make decisions without bothering everyone else.

Agreement in Faulty Systems

The problem was originally studied by Lamport et al. (1982) and is also known as the **Byzantine agreement problem,** referring to the numerous wars in which several armies needed to reach agreement on, for example, troop strengths while being faced with traitorous generals, conniving lieutenants, and so on. Consider the following solution, described in Lamport et al. (1982). In this case, we assume that processes are synchronous, messages are unicast while preserving ordering, and communication delay is bounded. We assume that there are N processes, where each process i will provide a value v_i to the others. The goal is let each process construct a vector V of length N, such that if process i is nonfaulty, $V[i] = v_i$. Otherwise, $V[i]$ is undefined. We assume that there are at most k faulty processes.

In Fig. 8-5 we illustrate the working of the algorithm for the case of $N = 4$ and $k = 1$. For these parameters, the algorithm operates in four steps. In step 1, every

nonfaulty process i sends v_i to every other process using reliable unicasting. Faulty processes may send anything. Moreover, because we are using multicasting, they may send different values to different processes. Let $v_i = i$. In Fig. 8-5(a) we see that process 1 reports 1, process 2 reports 2, process 3 lies to everyone, giving x, y, and z, respectively, and process 4 reports a value of 4. In step 2, the results of the announcements of step 1 are collected together in the form of the vectors of Fig. 8-5(b).



(a)

1 Got(1, 2, x, 4)
2 Got(1, 2, y, 4)
3 Got(1, 2, 3, 4)
4 Got(1, 2, z, 4)

(b)

1 Got	2 Got	4 Got
(1, 2, y, 4)	(1, 2, x, 4)	(1, 2, x, 4)
(a, b, c, d)	(e, f, g, h)	(1, 2, y, 4)
(1, 2, z, 4)	(1, 2, z, 4)	(i, j, k, l)

(c)

Figure 8-5. The Byzantine agreement problem for three nonfaulty and one faulty process. (a) Each process sends their value to the others. (b) The vectors that each process assembles based on (a). (c) The vectors that each process receives

Step 3 consists of every process passing its vector from Fig. 8-5(b) to every other process. In this way, every process gets three vectors, one from every other process. Here, too, process 3 lies, inventing 12 new values, *a* through l. The results of step 3 are shown in Fig. 8-5(c). Finally, in step 4, each process examines the ith element of each of the newly received vectors. If any value has a majority, that value is put into the result vector. If no value has a majority, the corresponding element of the result vector is marked *UNKNOWN*. From Fig. 8-5(c) we see that 1, 2, and 4 all come to agreement on the values for v_1, v_2, and v_4, which is the correct result. What these processes conclude regarding v_3 cannot be decided, but is also irrelevant. The goal of Byzantine agreement is that consensus is reached on the value for the nonfaulty processes only.

Now let us revisit this problem for $N = 3$ and $k = 1$, that is, only two nonfaulty process and one faulty one, as illustrated in Fig. 8-6. Here we see that in Fig. 8-6(c) neither of the correctly behaving processes sees a majority for element 1, element 2, or element 3, so all of them are marked *UNKNOWN*. The algorithm has failed to produce agreement.

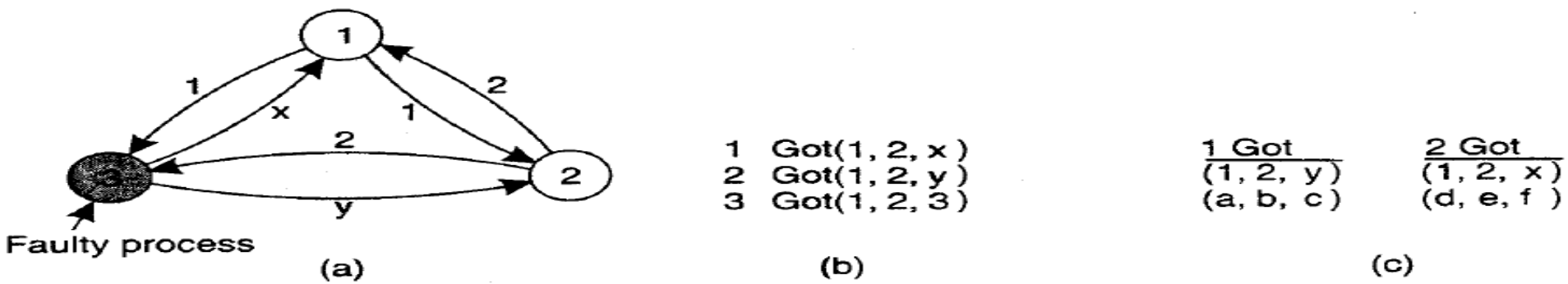


Figure 8-6. The same as Fig. 8-5, except now with two correct process and one faulty process.

Q: Consider a Web browser that returns an outdated cached page instead of a more recent one that had been updated at the server. Is this a failure, and if so, what kind of failure?

A: Whether or not it is a failure depends on the consistency that was promised to the user. If the browser promises to provide pages that are at most T time units old, it may exhibit performance failures. However, a browser can never live up to such a promise in the Internet. A weaker form of consistency is to provide one of the client-centric models discussed in Chap. 7. In that case, simply returning a page from the cache without checking its consistency may lead to a response failure.

DISTRIBUTED COMMIT

The atomic multicasting problem discussed in the previous section is an example of a more general problem, known as **distributed commit.** The distributed commit problem involves having an operation being performed by each member of a process group, or none at all.

Distributed commit is often established by means of a coordinator. In a simple scheme, this coordinator tells all other processes that are also involved, called participants, whether or not to (locally) perform the operation in question. This scheme is referred to as a **one-phase commit protocol.**

It has the obvious drawback that if one of the participants cannot actually perform the operation, there is no way to tell the coordinator. For example, in the case of distributed transactions, a local commit may not be possible because this would violate concurrency control constraints.

Two-Phase Commit

1. The coordinator sends a *VOTE_REQUEST* message to all participants.
2. When a participant receives a *VOTE_REQUEST* message, it returns either a *VOTE_COMMIT* message to the coordinator telling the coordinator that it is prepared to locally commit its part of the transaction, or otherwise a *VOTE_ABORT* message.
3. The coordinator collects all votes from the participants. If all participants have voted to commit the transaction, then so will the coordinator. In that case, it sends a *GLOBAL_COMMIT* message to all participants. However, if one participant had voted to abort the transaction, the coordinator will also decide to abort the transaction and multicasts a *GLOBAL_ABORT* message.
4. Each participant that voted for a commit waits for the final reaction by the coordinator. If a participant receives a *GLOBAL_COMMIT* message, it locally commits the transaction. Otherwise, when receiving a *GLOBAL_ABORT* message, the transaction is locally aborted as well.

Several problems arise when this basic 2PC protocol is used in a system where failures occur. First, note that the coordinator as well as the participants have states in which they block waiting for incoming messages.
Consequently, the protocol can easily fail when a process crashes for other processes may be indefinitely waiting for a message from that process. For this reason, timeout mechanism are used.

Actions by coordinator:

```
write START_2PC to local log;
multicast VOTE_REQUEST to all participants;
while not all votes have been collected {
        wait for any incoming vote;
        if timeout {
                write GLOBAL_ABORT to local log;
                multicast GLOBAL_ABORT to all participants;
                exit;
        }
        record vote;
}
if all participants sent VOTE_COMMIT and coordinator votes COMMIT {
        write GLOBAL_COMMIT to local log;
        multicast GLOBAL_COMMIT to all participants;
} else {
        write GLOBAL_ABORT to local log;
        multicast GLOBAL_ABORT to all participants;
}
```

Actions by participant:

```
write INIT to local log;
wait for VOTE_REQUEST from coordinator;
if timeout {
        write VOTE_ABORT to local log;
        exit;
}
if participant votes COMMIT {
        write VOTE_COMMIT to local log;
        send VOTE_COMMIT to coordinator;
        wait for DECISION from coordinator;
        if timeout {
                multicast DECISION_REQUEST to other participants;
                wait until DECISION is received; /* remain blocked */
                write DECISION to local log;
        }
        if DECISION == GLOBAL_COMMIT
                write GLOBAL_COMMIT to local log;
        else if DECISION == GLOBAL_ABORT
                write GLOBAL_ABORT to local log;
} else {
        write VOTE_ABORT to local log;
        send VOTE_ABORT to coordinator;
}
```

Actions for handling decision requests: /* executed by separate thread */

```
while true {
    wait until any incoming DECISION_REQUEST is received; /* remain blocked */
    read most recently recorded STATE from the local log;
    if STATE == GLOBAL_COMMIT
        send GLOBAL_COMMIT to requesting participant;
    else if STATE == INIT or STATE == GLOBAL_ABORT
        send GLOBAL_ABORT to requesting participant;
    else
        skip; /* participant remains blocked */
}
```

(b)

8.5.2 Three-Phase Commit

A problem with the two-phase commit protocol is that when the coordinator has crashed, participants may not be able to reach a final decision.
Consequently, participants may need to remain blocked until the coordinator recovers. Skeen (1981) developed a variant of 2PC, called the three-phase commit protocol (3PC), that avoids blocking processes in the presence of fail-stop crashes

1. There is no single state from which it is possible to make a transition directly to either a *COMMIT* or an *ABORT* state.
2. There is no state in which it is not possible to make a final decision, and from which a transition to a *COMMIT* state can be made.

The coordinator in 3PC starts with sending a *VOTE....REQUEST* message to all participants, after which it waits for incoming responses.
If any participant votes to abort the transaction, the final decision will be to abort as well, so the coordinator sends *GLOBAL-ABORT.*
However, when the transaction can be committed, a *PREPARE_COMMIT* message is sent. Only after each participant has acknowledged it is now prepared to commit, will the coordinator send the final *GLOBAL_COMMIT* message by which the transaction is actually committed.

Let's consider a specific banking problem where a customer wants to transfer funds between two accounts (checking and savings) held at a bank. The bank operates multiple branches, each maintaining its local database of customer accounts. The centralized database manages the customer accounts and processes transactions initiated by branches.

Scenario:

1. Prepare Phase:
    1. The centralized database (coordinator) receives a request from a branch to transfer funds between a checking account and a savings account.
    2. The coordinator initiates a transaction and sends a "prepare" message to all participating branches, asking them to prepare to commit the transaction.
    3. Each branch checks if it can successfully debit the source checking account and credit the destination savings account without violating any constraints (e.g., insufficient balance or transaction limits). If so, it replies with a "ready" message; otherwise, it replies with an "abort" message.

2. Commit Phase:

   If all branches respond with a "ready" message during the prepare phase, the coordinator sends a "commit" message to all participating branches, instructing them to commit the transaction.

   1. Upon receiving the "commit" message, each branch updates its local database to reflect the fund transfer (debiting the checking account and crediting the savings account) and acknowledges the coordinator with a "acknowledge" message.
   2. If any branch responds with an "abort" message during the prepare phase or if the coordinator detects a failure or timeout, it sends an "abort" message to all participating branches, instructing them to abort the transaction.
   3. Upon receiving the "abort" message, each branch rolls back any changes made during the prepare phase and acknowledges the coordinator with a "acknowledge" message.

Solution: To address potential challenges in this scenario, we can implement the following solutions:

1.Timeout Mechanism: Introduce timeouts at each phase to detect failures or network issues. If a branch or coordinator does not receive a response within the specified timeout, it can take appropriate action, such as aborting the transaction or initiating recovery procedures.

2.Logging and Recovery: Maintain transaction logs at each branch and the centralized database to record the state of transactions. Upon recovery from a failure, branches and the coordinator can replay logged transactions to restore consistency.

3.Coordinator Redundancy: Implement redundancy for the centralized database (coordinator) to ensure high availability. Use leader election or failover mechanisms to select a new coordinator in case of coordinator failure.

4.Participant Voting: Allow branches to vote on whether to commit or abort a transaction during the prepare phase. If a branch detects an issue (e.g., insufficient balance), it can vote to abort the transaction, preventing inconsistent updates.