Lecture 08: Fault Tolerance

Last update: October 30, 2019
Dependability

Basics

A component provides services to clients. To provide services, the component may require the services from other components ⇒ a component may depend on some other component.

Specifically

A component $C$ depends on $C^*$ if the correctness of $C$’s behavior depends on the correctness of $C^*$’s behavior. (Components are processes or channels.)
Dependability

Basics

A component provides services to clients. To provide services, the component may require the services from other components ⇒ a component may depend on some other component.

Specifically

A component $C$ depends on $C^*$ if the correctness of $C$’s behavior depends on the correctness of $C^*$’s behavior. (Components are processes or channels.)

Requirements related to dependability

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Availability</td>
<td>Readiness for usage</td>
</tr>
<tr>
<td>Reliability</td>
<td>Continuity of service delivery</td>
</tr>
<tr>
<td>Safety</td>
<td>Very low probability of catastrophes</td>
</tr>
<tr>
<td>Maintainability</td>
<td>How easy can a failed system be repaired</td>
</tr>
</tbody>
</table>
Reliability $R(t)$ of component $C$

Conditional probability that $C$ has been functioning correctly during $[0, t)$ given $C$ was functioning correctly at time $T = 0$.

Traditional metrics

- **Mean Time To Failure (MTTF):** The average time until a component fails.
- **Mean Time To Repair (MTTR):** The average time needed to repair a component.
- **Mean Time Between Failures (MTBF):** Simply $MTTF + MTTR$. 
Reliability versus availability

Availability $A(t)$ of component $C$

**Average fraction** of time that $C$ has been up-and-running in interval $[0, t)$.

- Long-term availability $A$: $A(\infty)$

**Note:** $A = \frac{MTTF}{MTBF} = \frac{MTTF}{MTTF + MTTR}$

Observation

Reliability and availability make sense only if we have an accurate notion of what a failure actually is.
# Terminology

## Failure, error, fault

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Failure</td>
<td>A component is not living up to its specifications</td>
<td>Crashed program</td>
</tr>
<tr>
<td>Error</td>
<td>Part of a component that can lead to a failure</td>
<td>Programming bug</td>
</tr>
<tr>
<td>Fault</td>
<td>Cause of an error</td>
<td>Sloppy programmer</td>
</tr>
</tbody>
</table>
## Terminology

### Handling faults

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fault prevention</td>
<td>Prevent the occurrence of a fault</td>
<td>Don’t hire sloppy programmers</td>
</tr>
<tr>
<td>Fault tolerance</td>
<td>Build a component such that it can mask the occurrence of a fault</td>
<td>Build each component by two independent programmers</td>
</tr>
<tr>
<td>Fault removal</td>
<td>Reduce the presence, number, or seriousness of a fault</td>
<td>Get rid of sloppy programmers</td>
</tr>
<tr>
<td>Fault forecasting</td>
<td>Estimate current presence, future incidence, and consequences of faults</td>
<td>Estimate how a recruiter is doing when it comes to hiring sloppy programmers</td>
</tr>
</tbody>
</table>
## Failure models

### Types of failures

<table>
<thead>
<tr>
<th>Type</th>
<th>Description of server’s behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>Crash failure</td>
<td>Halts, but is working correctly until it halts</td>
</tr>
<tr>
<td>Omission failure</td>
<td>Fails to respond to incoming requests</td>
</tr>
<tr>
<td></td>
<td>Fails to receive incoming messages</td>
</tr>
<tr>
<td></td>
<td>Fails to send messages</td>
</tr>
<tr>
<td>Timing failure</td>
<td>Response lies outside a specified time interval</td>
</tr>
<tr>
<td>Response failure</td>
<td>Response is incorrect</td>
</tr>
<tr>
<td></td>
<td>The value of the response is wrong</td>
</tr>
<tr>
<td></td>
<td>Deviates from the correct flow of control</td>
</tr>
<tr>
<td>Arbitrary failure</td>
<td>May produce arbitrary responses at arbitrary times</td>
</tr>
</tbody>
</table>
### Omission versus commission

Arbitrary failures are sometimes qualified as **malicious**. It is better to make the following distinction:

- **Omission failures**: a component fails to take an action that it should have taken
- **Commission failure**: a component takes an action that it should not have taken
Dependability versus security

Omission versus commission

Arbitrary failures are sometimes qualified as malicious. It is better to make the following distinction:

- **Omission failures**: a component fails to take an action that it should have taken
- **Commission failure**: a component takes an action that it should not have taken

Observation

Note that deliberate failures, be they omission or commission failures are typically security problems. Distinguishing between deliberate failures and unintentional ones is, in general, impossible.
Halting failures

Scenario

C no longer perceives any activity from $C^*$ — a halting failure? Distinguishing between a crash or omission/timing failure may be impossible.

Asynchronous versus synchronous systems

- **Asynchronous system**: no assumptions about process execution speeds or message delivery times → cannot reliably detect crash failures.

- **Synchronous system**: process execution speeds and message delivery times are bounded → we can reliably detect omission and timing failures.

- In practice we have partially synchronous systems: most of the time, we can assume the system to be synchronous, yet there is no bound on the time that a system is asynchronous → can normally reliably detect crash failures.
Halting failures

Assumptions we can make

<table>
<thead>
<tr>
<th>Halting type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fail-stop</td>
<td>Crash failures, but reliably detectable</td>
</tr>
<tr>
<td>Fail-noisy</td>
<td>Crash failures, eventually reliably detectable</td>
</tr>
<tr>
<td>Fail-silent</td>
<td>Omission or crash failures: clients cannot tell what went wrong</td>
</tr>
<tr>
<td>Fail-safe</td>
<td>Arbitrary, yet benign failures (i.e., they cannot do any harm)</td>
</tr>
<tr>
<td>Fail-arbitrary</td>
<td>Arbitrary, with malicious failures</td>
</tr>
</tbody>
</table>

Top-down

Severity: least → worst
Redundancy for failure masking

Types of redundancy

- **Information redundancy**: Add extra bits to data units so that errors can be recovered when bits are garbled.

- **Time redundancy**: Design a system such that an action can be performed again if anything went wrong. Typically used when faults are transient or intermittent.

- **Physical redundancy**: Add equipment or processes in order to allow one or more components to fail. This type is extensively used in distributed systems.
Process resilience

Basic idea

Protect against malfunctioning processes through process replication, organizing multiple processes into process group. Distinguish between flat groups and hierarchical groups.
Groups and failure masking

$k$-fault tolerant group

When a group can mask any $k$ concurrent member failures ($k$ is called degree of fault tolerance).
Groups and failure masking

**k-fault tolerant group**

When a group can mask any $k$ concurrent member failures ($k$ is called degree of fault tolerance).

**How large does a k-fault tolerant group need to be?**

- With **halting failures** (crash/omission/timing failures): we need a total of $k + 1$ members as no member will produce an incorrect result, so the result of one member is good enough.

- With **arbitrary failures**: we need $2k + 1$ members so that the correct result can be obtained through a majority vote.
Groups and failure masking

$k$-fault tolerant group

When a group can mask any $k$ concurrent member failures ($k$ is called degree of fault tolerance).

How large does a $k$-fault tolerant group need to be?

- With halting failures (crash/omission/timing failures): we need a total of $k + 1$ members as no member will produce an incorrect result, so the result of one member is good enough.
- With arbitrary failures: we need $2k + 1$ members so that the correct result can be obtained through a majority vote.

Important assumptions

- All members are identical
- All members process commands in the same order

Result: We can now be sure that all processes do exactly the same thing.
Consensus

Prerequisite

In a fault-tolerant process group, each nonfaulty process executes the same commands, and in the same order, as every other nonfaulty process.

Reformulation

Nonfaulty group members need to reach consensus on which command to execute next.
Realistic consensus: Paxos

Assumptions (rather weak ones, and realistic)

- A **partially synchronous** system (in fact, it may even be asynchronous).
- Communication between processes may be **unreliable**: messages may be lost, duplicated, or reordered.
- Corrupted message can be **detected** (and thus subsequently ignored).
- All operations are **deterministic**: once an execution is started, it is known exactly what it will do.
- Processes may exhibit **crash failures**, but **not arbitrary failures**.
- Processes **do not collude**.

Understanding Paxos

We will build up Paxos from scratch to understand where many consensus algorithms actually come from.
Paxos essentials

Starting point

- We assume a client-server configuration, with initially one primary server.
- To make the server more robust, we start with adding a backup server.
- To ensure that all commands are executed in the same order at both servers, the primary assigns unique sequence numbers to all commands. In Paxos, the primary is called the leader.
- Assume that actual commands can always be restored (either from clients or servers) ⇒ we consider only control messages.
Observation

Paxos needs at least three servers
Required number of servers

Observation
Paxos needs at least three servers

Adapted fundamental rule
In Paxos with three servers, a server $S$ cannot execute an operation $o$ until it has received at least one (other) $\text{LEARN}(o)$ message, so that it knows that a majority of servers will execute $o$. 
Why having **3k** processes is not enough

Faulty process

First message round

Second message round
Why having $3k + 1$ processes is enough
Consistency, availability, and partitioning

CAP theorem

Any networked system providing shared data can provide only two of the following three properties:

- **C**: consistency, by which a shared and replicated data item appears as a single, up-to-date copy
- **A**: availability, by which updates will always be eventually executed
- **P**: Tolerant to the partitioning of process group.

Conclusion

In a network subject to communication failures, it is impossible to realize an atomic read/write shared memory that guarantees a response to every request.
Failure detection

Issue
How can we reliably detect that a process has actually crashed?

General model
- Each process is equipped with a failure detection module
- A process $P$ probes another process $Q$ for a reaction
- If $Q$ reacts: $Q$ is considered to be alive (by $P$)
- If $Q$ does not react with $t$ time units: $Q$ is suspected to have crashed

Observation for a synchronous system
a suspected crash $\equiv$ a known crash
Practical failure detection

Implementation

- If $P$ did not receive heartbeat from $Q$ within time $t$: $P$ suspects $Q$.
- If $Q$ later sends a message (which is received by $P$):
  - $P$ stops suspecting $Q$
  - $P$ increases the timeout value $t$
- **Note**: if $Q$ did crash, $P$ will keep suspecting $Q$. 
Reliable RPC: lost reply messages

The real issue

What the client notices, is that it is not getting an answer. However, it cannot decide whether this is caused by a lost request, a crashed server, or a lost response.

Partial solution

Design the server such that its operations are idempotent: repeating the same operation is the same as carrying it out exactly once:

- pure read operations
- strict overwrite operations

Many operations are inherently nonidempotent, such as many banking transactions.
Reliable RPC: client crash

Problem
The server is doing work and holding resources for nothing (called doing an orphan computation).

Solution
- Orphan is killed (or rolled back) by the client when it recovers
- Client broadcasts new epoch number when recovering $\Rightarrow$ server kills client’s orphans
- Require computations to complete in a $T$ time units. Old ones are simply removed.
Distributed commit protocols

Problem

Have an operation being performed by each member of a process group, or none at all.

- **Reliable multicasting**: a message is to be delivered to all recipients.
- **Distributed transaction**: each local transaction must succeed.
Two-phase commit protocol (2PC)

Essence

The client who initiated the computation acts as **coordinator**; processes required to commit are the **participants**.

- **Phase 1a**: Coordinator sends **VOTE-REQUEST** to participants (also called a **pre-write**)

- **Phase 1b**: When participant receives **VOTE-REQUEST** it returns either **VOTE-COMMIT** or **VOTE-ABORT** to coordinator. If it sends **VOTE-ABORT**, it aborts its local computation

- **Phase 2a**: Coordinator collects all votes; if all are **VOTE-COMMIT**, it sends **GLOBAL-COMMIT** to all participants, otherwise it sends **GLOBAL-ABORT**

- **Phase 2b**: Each participant waits for **GLOBAL-COMMIT** or **GLOBAL-ABORT** and handles accordingly.
2PC - Finite state machines

Coordinator

Participant
Recovery: Background

Essence
When a failure occurs, we need to bring the system into an error-free state:

- **Forward error recovery**: Find a new state from which the system can continue operation
- **Backward error recovery**: Bring the system back into a previous error-free state

Practice
Use backward error recovery, requiring that we establish recovery points

Observation
Recovery in distributed systems is complicated by the fact that processes need to cooperate in identifying a consistent state from where to recover
Consistent recovery state

Requirement
Every message that has been received is also shown to have been sent in the state of the sender.

Recovery line
Assuming processes regularly checkpoint their state, the most recent consistent global checkpoint.
Cascaded rollback

Observation

If checkpointing is done at the “wrong” instants, the recovery line may lie at system startup time. We have a so-called cascaded rollback.

Could you draw a valid recovery line rather than startup?