Distributed Systems

- 48 - Communication Basics
- 49 - NFS
- 50 - AFS
- GFS
- Fault Tolerance

Distributed GFS

GFS master

Legend:
- Data messages
- Control messages

Application

GFS client

(file name, chunk index)

(chunk handle, chunk locations)

(chunk handle, byte range)

chunk data
Distributed GFS

GFS master

consensus group is fault-tolerant; any one of the three can fail without halting the entire system

Fault Tolerance dependability

- 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.)

<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>
Fault Tolerance \textit{basics}

- 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):
    - average time until a component fails.
  - Mean Time To Repair (MTTR):
    - average time needed to repair a component.
  - Mean Time Between Failures (MTBF)
    - Simply MTTF + MTTR.

Reliability vs 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: $A = \frac{MTTF}{MTBF} = \frac{MTTF}{MTTF+MTTR}$

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

<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>

## 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

<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 (<em>fail stop</em>)</td>
</tr>
<tr>
<td>Omission failure</td>
<td></td>
</tr>
</tbody>
</table>
  *receive omission*
  *send omission* | Fails to respond to incoming requests  
  Fails to receive incoming messages  
  Fails to send messages                                      |
| Timing failure | Response lies outside a specified time interval                                                    |
| Response failure | 
  *Value failure*
  *State-transition failure* | Response is incorrect  
  The value of the response is wrong  
  Deviates from the correct flow of control                      |
| Arbitrary failure | May produce arbitrary responses at arbitrary times                                                |

### Dependability vs Security

#### Omission versus commission

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

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

#### Observation

*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 is difficult to impossible.

Asynchronous versus synchronous systems

- **Asynchronous system**: no assumptions about process execution speeds or message delivery times \( \rightarrow \) cannot reliably detect crash failures.
- **Synchronous system**: process execution speeds and message delivery times are bounded \( \rightarrow \) 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 \( \rightarrow \) can normally reliably detect crash failures.

<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>
Process Resilience

Basic idea
Protect against malfunctioning processes through *process replication*, organizing multiple processes into a *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*).

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. If *k* fail silently, the answer of the other can be used.
- With *arbitrary failures*: we need 2k + 1 members so that the correct result can be obtained through a majority vote. Up to *k* could be malicious (lie, prevaricate), so we need *k*+1 who agree to reach consensus. If at most fail, there should be *n*+1 correct servers left.

Important assumptions:
- All members are identical
- All members process commands in the same order

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

Prerequisite
In a fault-tolerant process group, each nonfaulty process commits 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 commit next.

Motivating Paxos by looking at consensus

Assumptions (rather weak ones, and realistic)
- System is partially synchronous (may even be asynchronous).
- Communication between processes may be unreliable:
  - messages may be lost, duplicated, or reordered.
- Corrupted messages can be detected
  - and thus subsequently ignored
- All values 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 to Paxos by looking at problems that occur.
Two Servers \textit{leader + backup}

- The leader sends an \textit{accept} message $\text{ACCEPT}(o, t)$ to backups when assigning a timestamp $t$ to command $o$.

Two Servers \textit{and a crash!}

Problem

Servers have diverged because primary crashes \textit{after executing} an value, but the backup \textit{never received} the accept message.