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 versus availability

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>
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

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.
Fault tolerance: Introduction to fault tolerance

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**.
## 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>
Redundancy for failure masking

Types of redundancy

- **Information redundancy**: Add extra bits to data units so that errors can 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.
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.
Flooding-based consensus

System model

- A process group \( P = \{P_1, \ldots, P_n\} \)
- **Fail-stop** failure semantics, i.e., with *reliable failure detection*
- A client contacts a \( P_i \) requesting it to execute a command
- Every \( P_i \) maintains a list of proposed commands
Flooding-based consensus

System model

- A process group $P = \{P_1, \ldots, P_n\}$
- Fail-stop failure semantics, i.e., with reliable failure detection
- A client contacts a $P_i$ requesting it to execute a command
- Every $P_i$ maintains a list of proposed commands

Basic algorithm (based on rounds)

1. In round $r$, $P_i$ multicasts its known set of commands $C^r_i$ to all others
2. At the end of $r$, each $P_i$ merges all received commands into a new $C^{r+1}_i$.
3. Next command $cmd_i$ selected through a globally shared, deterministic function: $cmd_i \leftarrow select(C^{r+1}_i)$. 
Flooding-based consensus: Example

Observations

- $P_2$ received all proposed commands from all other processes $\Rightarrow$ makes decision.

- $P_3$ may have detected that $P_1$ crashed, but does not know if $P_2$ received anything, i.e., $P_3$ cannot know if it has the same information as $P_2$ $\Rightarrow$ cannot make decision (same for $P_4$).
Realizing fault tolerance

Observation
Considering that the members in a fault-tolerant process group are so tightly coupled, we may bump into considerable performance problems, but perhaps even situations in which realizing fault tolerance is impossible.

Question
Are there limitations to what can be readily achieved?
- What is needed to enable reaching consensus?
- What happens when groups are partitioned?
Distributed consensus: when can it be reached

Formal requirements for consensus
- Processes produce the same output value
- Every output value must be valid
- Every process must eventually provide output
Failure detection

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 $Q$. 
Point-to-point communication

- reliable point-to-point communication is established using a reliable transport protocol, such as TCP.
- TCP masks omission failures by using ACKs and retransmissions.
- Crash failures are not masked.
A message sent to a process group $G$ should be delivered to each member of $G$. **Important**: make distinction between receiving and delivering messages.
Less simple reliable group communication

Reliable communication in the presence of faulty processes

Group communication is reliable when it can be guaranteed that a message is received and subsequently delivered by all nonfaulty group members.

Tricky part

Agreement is needed on what the group actually looks like before a received message can be delivered.
Simple reliable group communication

Reliable communication, but assume nonfaulty processes

Reliable group communication now boils down to reliable multicasting: is a message received and delivered to each recipient, as intended by the sender.
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
2PC – Failing participant

Analysis: participant crashes in state $S$, and recovers to $S$

- **INIT**: decide to abort and informs the coordinator

Observation: When distributed commit is required, having participants use temporary workspaces to keep their results allows for simple recovery in the presence of failures.
Analysis: participant crashes in state S, and recovers to S

- **READY**: Participant is waiting to either commit or abort. After recovery, participant needs to know which state transition it should make $\Rightarrow$ contact other process
2PC – Failing participant

Analysis: participant crashes in state S, and recovers to S

- **ABORT**: Merely make entry into abort state idempotent, e.g., removing the workspace of results
2PC – Failing participant

Analysis: participant crashes in state $S$, and recovers to $S$

- **COMMIT**: Also make entry into commit state idempotent, e.g., copying workspace to storage.
2PC – Failing participant

Analysis: participant crashes in state $S$, and recovers to $S$

- **INIT**: decide to abort and informs the coordinator
- **READY**: Participant is waiting to either commit or abort. After recovery, participant needs to know which state transition it should make $\Rightarrow$ contact other process
- **ABORT**: Merely make entry into abort state idempotent, e.g., removing the workspace of results
- **COMMIT**: Also make entry into commit state idempotent, e.g., copying workspace to storage.

Observation
When distributed commit is required, having participants use temporary workspaces to keep their results allows for simple recovery in the presence of failures.
Fault tolerance: Distributed commit

2PC – Failing participant

When a recovery is needed to *READY* state, check state of other participants.

Recovering participant *P* contacts another participant *Q*

<table>
<thead>
<tr>
<th>State of <em>Q</em></th>
<th>Action by <em>P</em></th>
</tr>
</thead>
<tbody>
<tr>
<td>COMMIT</td>
<td>Make transition to COMMIT</td>
</tr>
<tr>
<td>ABORT</td>
<td>Make transition to ABORT</td>
</tr>
<tr>
<td>INIT</td>
<td>Make transition to ABORT</td>
</tr>
<tr>
<td>READY</td>
<td>Contact another participant</td>
</tr>
</tbody>
</table>

Result

If all participants are in the *READY* state, the protocol blocks. Apparently, the coordinator is failing. **Note:** The protocol prescribes that we need the decision from the coordinator.
Observation
The real problem lies in the fact that the coordinator’s final decision may not be available for some time (or actually lost).

Alternative
Let a participant $P$ in the \textit{READY} state timeout when it hasn’t received the coordinator’s decision; $P$ tries to find out what other participants know (as discussed).

Observation
Essence of the problem is that a recovering participant cannot make a local decision: it is dependent on other (possibly failed) processes.
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.
Coordinated checkpointing

Essence
Each process takes a checkpoint after a globally coordinated action.

Simple solution
Use a two-phase blocking protocol:
Coordinated checkpointing

Essence
Each process takes a checkpoint after a globally coordinated action.

Simple solution
Use a two-phase blocking protocol:
- A coordinator multicasts a checkpoint request message
Coordinated checkpointing

Essence
Each process takes a checkpoint after a globally coordinated action.

Simple solution
Use a two-phase blocking protocol:
- A coordinator multicasts a **checkpoint request** message
- When a participant receives such a message, it takes a checkpoint, stops sending (application) messages, and reports back that it has taken a checkpoint
Coordinated checkpointing

Essence

Each process takes a checkpoint after a globally coordinated action.

Simple solution

Use a two-phase blocking protocol:

- A coordinator multicasts a `checkpoint request` message
- When a participant receives such a message, it takes a checkpoint, stops sending (application) messages, and reports back that it has taken a checkpoint
- When all checkpoints have been confirmed at the coordinator, the latter broadcasts a `checkpoint done` message to allow all processes to continue
Coordinated checkpointing

**Essence**
Each process takes a checkpoint after a globally coordinated action.

**Simple solution**
Use a two-phase blocking protocol:
- A coordinator multicasts a `checkpoint request` message
- When a participant receives such a message, it takes a checkpoint, stops sending (application) messages, and reports back that it has taken a checkpoint
- When all checkpoints have been confirmed at the coordinator, the latter broadcasts a `checkpoint done` message to allow all processes to continue

**Observation**
It is possible to consider only those processes that depend on the recovery of the coordinator, and ignore the rest
Message logging

Alternative

Instead of taking an (expensive) checkpoint, try to *replay* your (communication) behavior from the most recent checkpoint ⇒ store messages in a log.

Assumption

We assume a *piecewise deterministic* execution model:

- The execution of each process can be considered as a sequence of state intervals
- Each state interval starts with a nondeterministic event (e.g., message receipt)
- Execution in a state interval is deterministic

Conclusion

If we record nondeterministic events (to replay them later), we obtain a deterministic execution model that will allow us to do a complete replay.