### 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

<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
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>
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.
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).

- 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.
**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 2*k* + 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 \( C_i^r \) to all others
2. At the end of \( r \), each \( P_i \) merges all received commands into a new \( C_i^{r+1} \).
3. Next command \( cmd_i \) selected through a globally shared, deterministic function: \( cmd_i \leftarrow \text{select}(C_i^{r+1}) \).
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 $P_4$).
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.
Two-server situation
Fault tolerance: Process resilience

Example: Paxos

Handling lost messages

Some Paxos terminology

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

- A backup responds by sending a learn message: $\text{LEARN}(o, t)$

- When the leader notices that operation $o$ has not yet been learned, it retransmits $\text{ACCEPT}(o, t)$ with the original timestamp.
Two servers and one crash: problem

Problem
Primary crashes after executing an operation, but the backup never received the accept message.
Two servers and one crash: solution

Solution

Never execute an operation before it is clear that is has been learned.
Three servers and two crashes: still a problem?

Example: Paxos

Leader

Understanding Paxos
Three servers and two crashes: still a problem?

Scenario

What happens when \texttt{LEARN}(o^1) as sent by S_2 to S_1 is lost?
Three servers and two crashes: still a problem?

Scenario

What happens when LEARN(o^1) as sent by S_2 to S_1 is lost?

Solution

S_2 will also have to wait until it knows that S_3 has learned o^1.
Paxos: fundamental rule

In Paxos, a server $S$ cannot execute an operation $o$ until it has received a $\text{LEARN}(o)$ from all other nonfaulty servers.
Failure detection

Practice

Reliable failure detection is practically impossible. A solution is to set timeouts, but take into account that a detected failure may be false.
Failure detection

Practice

Reliable failure detection is practically impossible. A solution is to set timeouts, but take into account that a detected failure may be false.

Understanding Paxos
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 $o$. 
Required number of servers

Assumptions before taking the next steps

- Initially, $S_1$ is the leader.
- A server can reliably detect it has missed a message, and recover from that miss.
- When a new leader needs to be elected, the remaining servers follow a strictly deterministic algorithm, such as $S_1 \rightarrow S_2 \rightarrow S_3$.
- A client cannot be asked to help the servers to resolve a situation.
Required number of servers

Assumptions before taking the next steps

- Initially, $S_1$ is the leader.
- A server can reliably detect it has missed a message, and recover from that miss.
- When a new leader needs to be elected, the remaining servers follow a strictly deterministic algorithm, such as $S_1 \rightarrow S_2 \rightarrow S_3$.
- A client cannot be asked to help the servers to resolve a situation.

Observation

If either one of the backups ($S_2$ or $S_3$) crashes, Paxos will behave correctly: operations at nonfaulty servers are executed in the same order.
Leader crashes after executing $o^1$
Leader crashes after executing $o^1$

$S_3$ is completely ignorant of any activity by $S_1$

- $S_2$ received $\text{ACCEPT}(o, 1)$, detects crash, and becomes leader.
- $S_3$ even never received $\text{ACCEPT}(o, 1)$.
- $S_2$ sends $\text{ACCEPT}(o^2, 2) \Rightarrow S_3$ sees unexpected timestamp and tells $S_2$ that it missed $o^1$.
- $S_2$ retransmits $\text{ACCEPT}(o^1, 1)$, allowing $S_3$ to catch up.
Leader crashes after executing $o^1$

$S_3$ is completely ignorant of any activity by $S_1$

- $S_2$ received $\text{ACCEPT}(o, 1)$, detects crash, and becomes leader.
- $S_3$ even never received $\text{ACCEPT}(o, 1)$.
- $S_2$ sends $\text{ACCEPT}(o^2, 2) \Rightarrow S_3$ sees unexpected timestamp and tells $S_2$ that it missed $o^1$.
- $S_2$ retransmits $\text{ACCEPT}(o^1, 1)$, allowing $S_3$ to catch up.

$S_2$ missed $\text{ACCEPT}(o^1, 1)$

- $S_2$ did detect crash and became new leader
- $S_2$ sends $\text{ACCEPT}(o^1, 1) \Rightarrow S_3$ retransmits $\text{LEARN}(o^1)$.
- $S_2$ sends $\text{ACCEPT}(o^2, 1) \Rightarrow S_3$ tells $S_2$ that it apparently missed $\text{ACCEPT}(o^1, 1)$ from $S_1$, so that $S_2$ can catch up.
Leader crashes after sending \texttt{ACCEPT}(o^1, 1)

\textbf{S_3 is completely ignorant of any activity by S_1}

As soon as \textit{S_2} announces that \textit{o}^2 is to be accepted, \textit{S_3} will notice that it missed an operation and can ask \textit{S_2} to help recover.

\textbf{S_2 had missed \texttt{ACCEPT}(o^1, 1)}

As soon as \textit{S_2} proposes an operation, it will be using a stale timestamp, allowing \textit{S_3} to tell \textit{S_2} that it missed operation \textit{o}^1.
Leader crashes after sending $\text{ACCEPT}(o^1, 1)$

$S_3$ is completely ignorant of any activity by $S_1$

As soon as $S_2$ announces that $o^2$ is to be accepted, $S_3$ will notice that it missed an operation and can ask $S_2$ to help recover.

$S_2$ had missed $\text{ACCEPT}(o^1, 1)$

As soon as $S_2$ proposes an operation, it will be using a stale timestamp, allowing $S_3$ to tell $S_2$ that it missed operation $o^1$.

Observation

Paxos (with three servers) behaves correctly when a single server crashes, regardless when that crash took place.
False crash detections

Problem and solution

$S_3$ receives $\text{ACCEPT}(o^1, 1)$, but much later than $\text{ACCEPT}(o^2, 1)$. If it knew who the current leader was, it could safely reject the delayed accept message. Therefore, leaders should include their ID in messages.
But what about progress?

When $S_2$ takes over, it needs to make sure that any outstanding operations initiated by $S_1$ have been properly flushed, i.e., executed by enough servers. This requires an explicit leadership takeover by which other servers are informed before sending out new accept messages.
But what about progress?

Essence of solution

When $S_2$ takes over, it needs to make sure that any outstanding operations initiated by $S_1$ have been properly flushed, i.e., executed by enough servers. This requires an explicit leadership takeover by which other servers are informed before sending out new accept messages.
Consensus under arbitrary failure semantics

Essence

We consider process groups in which communication between process is inconsistent: (a) improper forwarding of messages, or (b) telling different things to different processes.
Consensus under arbitrary failure semantics

System model
- We consider a primary $P$ and $n-1$ backups $B_1, \ldots, B_{n-1}$.
- A client sends $v \in \{T, F\}$ to $P$.
- Messages may be lost, but this can be detected.
- Messages cannot be corrupted beyond detection.
- A receiver of a message can reliably detect its sender.

Byzantine agreement: requirements

BA1: Every nonfaulty backup process stores the same value.
BA2: If the primary is nonfaulty then every nonfaulty backup process stores exactly what the primary had sent.

Observation
- Primary faulty $\Rightarrow$ BA1 says that backups may store the same, but different (and thus wrong) value than originally sent by the client.
- Primary not faulty $\Rightarrow$ satisfying BA2 implies that BA1 is satisfied.
Why having $3k$ processes is not enough
Why having $3k + 1$ processes is enough
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

## Message ordering

<table>
<thead>
<tr>
<th>Process behavior</th>
<th>Message transmission</th>
<th>Communication delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synchronous</td>
<td>Unicast</td>
<td>Multicast</td>
</tr>
<tr>
<td>Asynchronous</td>
<td>Unordered</td>
<td>Ordered</td>
</tr>
</tbody>
</table>

## Unicast

- X

## Multicast

- X

## Formal requirements for consensus

- Processes produce the same output value
- Every output value must be valid
- Every process must eventually provide output
Fault tolerance: Process resilience

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 remote procedure calls

What can go wrong?

1. The client is unable to locate the server.
2. The request message from the client to the server is lost.
3. The server crashes after receiving a request.
4. The reply message from the server to the client is lost.
5. The client crashes after sending a request.
Reliable remote procedure calls

What can go wrong?

1. The client is unable to locate the server.
2. The request message from the client to the server is lost.
3. The server crashes after receiving a request.
4. The reply message from the server to the client is lost.
5. The client crashes after sending a request.

Two “easy” solutions

1. (cannot locate server): just report back to client
2. (request was lost): just resend message
Reliable RPC: server crash

(a) (b) (c)

Problem

Where (a) is the normal case, situations (b) and (c) require different solutions. However, we don’t know what happened. Two approaches:

- **At-least-once-semantics**: The server guarantees it will carry out an operation at least once, no matter what.

- **At-most-once-semantics**: The server guarantees it will carry out an operation at most once.
Fault tolerance: Reliable client-server communication
RPC semantics in the presence of failures

Why fully transparent server recovery is impossible

Three type of events at the server
(Assume the server is requested to update a document.)

- **M**: send the completion message
- **P**: complete the processing of the document
- **C**: crash

Six possible orderings
(Actions between brackets never take place)

1. \( M \rightarrow P \rightarrow C \): Crash after reporting completion.
2. \( M \rightarrow C \rightarrow P \): Crash after reporting completion, but before the update.
3. \( P \rightarrow M \rightarrow C \): Crash after reporting completion, and after the update.
4. \( P \rightarrow C (\rightarrow M) \): Update took place, and then a crash.
5. \( C (\rightarrow P \rightarrow M) \): Crash before doing anything
6. \( C (\rightarrow M \rightarrow P) \): Crash before doing anything

Server crashes
## Why fully transparent server recovery is impossible

<table>
<thead>
<tr>
<th>Reissue strategy</th>
<th>Strategy M → P</th>
<th>Strategy P → M</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MPC</td>
<td>MC(P)</td>
</tr>
<tr>
<td>Always</td>
<td>DUP</td>
<td>OK</td>
</tr>
<tr>
<td>Never</td>
<td>OK</td>
<td>ZERO</td>
</tr>
<tr>
<td>Only when ACKed</td>
<td>DUP</td>
<td>OK</td>
</tr>
<tr>
<td>Only when not ACKed</td>
<td>OK</td>
<td>ZERO</td>
</tr>
</tbody>
</table>

- **OK** = Document updated once
- **DUP** = Document updated twice
- **ZERO** = Document not updated at all

---

### Notes

- Server crashes

---

**Fault tolerance:** Reliable client-server communication

**RPC semantics in the presence of failures**

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 ⇒ server kills client’s orphans
- Require computations to **complete in a \( T \) time units**. Old ones are simply removed.
Simple reliable group communication

Intuition

A message sent to a process group $G$ should be delivered to each member of $G$. **Important**: make distinction between receiving and delivering messages.
Fault tolerance: Reliable group communication

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.
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.
Fault tolerance: Distributed commit

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.
Fault tolerance: Distributed commit

2PC - Finite state machines

Coordinator

Participant
2PC – Failing participant

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

- **INIT**: No problem: participant was unaware of protocol

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 $\Rightarrow$ log the coordinator’s decision
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**: No problem: participant was unaware of protocol

- **READY**: Participant is waiting to either commit or abort. After recovery, participant needs to know which state transition it should make $\Rightarrow$ log the coordinator’s decision

- **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

Alternative

When a recovery is needed to \textit{READY} state, check state of other participants $\Rightarrow$ no need to log coordinator’s decision.

Recovering participant $P$ contacts another participant $Q$

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

Result

If all participants are in the \textit{READY} state, the protocol blocks. Apparently, the coordinator is failing. \textbf{Note:} The protocol prescribes that we need the decision from the coordinator.
2PC – Failing 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 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.
Coordinator in Python

```python
class Coordinator:
    def run(self):
        yetToReceive = list(participants)
        self.log.info('WAIT')
        self.chan.sendTo(participants, VOTE_REQUEST)

        while len(yetToReceive) > 0:
            msg = self.chan.recvFrom(participants, TIMEOUT)
            if (not msg) or (msg[1] == VOTE_ABORT):
                self.log.info('ABORT')
                self.chan.sendTo(participants, GLOBAL_ABORT)
                return
            else: # msg[1] == VOTE_COMMIT
                yetToReceive.remove(msg[0])
                self.log.info('COMMIT')
                self.chan.sendTo(participants, GLOBAL_COMMIT)
```

class Participant:
    def run(self):
        msg = self.chan.recvFrom(coordinator, TIMEOUT)
        if not msg:
            # Crashed coordinator - give up entirely
            decision = LOCAL_ABORT
        else:
            # Coordinator will have sent VOTE_REQUEST
            decision = self.do_work()
            if decision == LOCAL_ABORT:
                self.chan.sendTo(coordinator, VOTE_ABORT)
            else:
                # Ready to commit, enter READY state
                self.chan.sendTo(coordinator, VOTE_COMMIT)
                msg = self.chan.recvFrom(coordinator, TIMEOUT)
        if not msg:
            # Crashed coordinator - check the others
            self.chan.sendTo(all_participants, NEED_DECISION)
            while True:
                msg = self.chan.recvFromAny()
                if msg[1] in [GLOBAL_COMMIT, GLOBAL_ABORT, LOCAL_ABORT]:
                    decision = msg[1]
                    break
        else:
            # Coordinator came to a decision
            decision = msg[1]

while True:
    # Help any other participant when coordinator crashed
    msg = self.chan.recvFrom(all_participants)
    if msg[1] == NEED_DECISION:
        self.chan.sendTo([msg[0]], decision)
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
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.
Independent checkpointing

Essence

Each process independently takes checkpoints, with the risk of a cascaded rollback to system startup.
Independent checkpointing

Essence

Each process independently takes checkpoints, with the risk of a cascaded rollback to system startup.

- Let $CP_i(m)$ denote $m^{th}$ checkpoint of process $P_i$ and $INT_i(m)$ the interval between $CP_i(m-1)$ and $CP_i(m)$. 

Observation

If process $P_i$ rolls back to $CP_i(m-1)$, $P_j$ must roll back to $CP_j(n-1)$. 

Independent checkpointing

**Essence**

Each process independently takes checkpoints, with the risk of a cascaded rollback to system startup.

- Let $CP_i(m)$ denote the $m^{th}$ checkpoint of process $P_i$ and $INT_i(m)$ the interval between $CP_i(m - 1)$ and $CP_i(m)$.
- When process $P_i$ sends a message in interval $INT_i(m)$, it piggybacks $(i, m)$.
Independent checkpointing

Essence

Each process independently takes checkpoints, with the risk of a cascaded rollback to system startup.

- Let $CP_i(m)$ denote $m^{th}$ checkpoint of process $P_i$ and $INT_i(m)$ the interval between $CP_i(m-1)$ and $CP_i(m)$.
- When process $P_i$ sends a message in interval $INT_i(m)$, it piggybacks $(i, m)$
- When process $P_j$ receives a message in interval $INT_j(n)$, it records the dependency $INT_i(m) \rightarrow INT_j(n)$. 
Independent checkpointing

Essence

Each process independently takes checkpoints, with the risk of a cascaded rollback to system startup.

- Let $CP_i(m)$ denote $m^{th}$ checkpoint of process $P_i$ and $INT_i(m)$ the interval between $CP_i(m-1)$ and $CP_i(m)$.

- When process $P_i$ sends a message in interval $INT_i(m)$, it piggybacks $(i, m)$

- When process $P_j$ receives a message in interval $INT_j(n)$, it records the dependency $INT_i(m) \rightarrow INT_j(n)$.

- The dependency $INT_i(m) \rightarrow INT_j(n)$ is saved to storage when taking checkpoint $CP_j(n)$. 
Independent checkpointing

**Essence**

Each process independently takes checkpoints, with the risk of a cascaded rollback to system startup.

- Let $CP_i(m)$ denote $m^{th}$ checkpoint of process $P_i$ and $INT_i(m)$ the interval between $CP_i(m-1)$ and $CP_i(m)$.
- When process $P_i$ sends a message in interval $INT_i(m)$, it piggybacks $(i, m)$
- When process $P_j$ receives a message in interval $INT_j(n)$, it records the dependency $INT_i(m) \rightarrow INT_j(n)$.
- The dependency $INT_i(m) \rightarrow INT_j(n)$ is saved to storage when taking checkpoint $CP_j(n)$.

**Observation**

If process $P_i$ rolls back to $CP_i(m-1)$, $P_j$ must roll back to $CP_j(n-1)$.
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.
Message logging and consistency

When should we actually log messages?

Avoid orphan processes:

- Process Q has just received and delivered messages \( m_1 \) and \( m_2 \)
- Assume that \( m_2 \) is never logged.
- After delivering \( m_1 \) and \( m_2 \), Q sends message \( m_3 \) to process R
- Process R receives and subsequently delivers \( m_3 \): it is an orphan.

![Diagram showing message logging and consistency](image)
Message-logging schemes

**Notations**

- $\text{DEP}(m)$: processes to which $m$ has been delivered. If message $m^*$ is causally dependent on the delivery of $m$, and $m^*$ has been delivered to $Q$, then $Q \in \text{DEP}(m)$.

- $\text{COPY}(m)$: processes that have a copy of $m$, but have not (yet) reliably stored it.

- $\text{FAIL}$: the collection of crashed processes.

**Characterization**

$Q$ is orphaned $\iff \exists m : Q \in \text{DEP}(m)$ and $\text{COPY}(m) \subseteq \text{FAIL}$
Message-logging schemes

Pessimistic protocol

For each nonstable message $m$, there is at most one process dependent on $m$, that is $|\text{DEP}(m)| \leq 1$.

Consequence

An unstable message in a pessimistic protocol must be made stable before sending a next message.
Message-logging schemes

Optimistic protocol
For each unstable message \( m \), we ensure that if \( \text{COPY}(m) \subseteq \text{FAIL} \), then eventually also \( \text{DEP}(m) \subseteq \text{FAIL} \).

Consequence
To guarantee that \( \text{DEP}(m) \subseteq \text{FAIL} \), we generally rollback each orphan process \( Q \) until \( Q \not\in \text{DEP}(m) \).