

# **Distributed Systems**

(4th edition, version 01)

## **Chapter 08: Fault Tolerance**

# Dependability

## Basics

A **component** provides **services** to **clients**. To provide services, the component may require the services from other components  $\Rightarrow$  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  $\Rightarrow$  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

Requirement	Description
Availability	Readiness for usage
Reliability	Continuity of service delivery
Safety	Very low probability of catastrophes
Maintainability	How easy can a failed system be repaired

# 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

Term	Description	Example
Failure	A component is not living up to its specifications	Crashed program
Error	Part of a component that can lead to a failure	Programming bug
Fault	Cause of an error	Sloppy programmer

# Terminology

## Handling faults

Term	Description	Example
Fault prevention	Prevent the occurrence of a fault	Don't hire sloppy programmers
Fault tolerance	Build a component such that it can mask the occurrence of a fault	Build each component by two independent programmers
Fault removal	Reduce the presence, number, or seriousness of a fault	Get rid of sloppy programmers
Fault forecasting	Estimate current presence, future incidence, and consequences of faults	Estimate how a recruiter is doing when it comes to hiring sloppy programmers

# Failure models

## Types of failures

Type	Description of server's behavior
<b>Crash failure</b>	Halts, but is working correctly until it halts
<b>Omission failure</b> <i>Receive omission</i> <i>Send omission</i>	Fails to respond to incoming requests Fails to receive incoming messages Fails to send messages
<b>Timing failure</b>	Response lies outside a specified time interval
<b>Response failure</b> <i>Value failure</i> <i>State-transition failure</i>	Response is incorrect The value of the response is wrong Deviates from the correct flow of control
<b>Arbitrary failure</b>	May produce arbitrary responses at arbitrary times



# 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

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

# Redundancy for failure masking

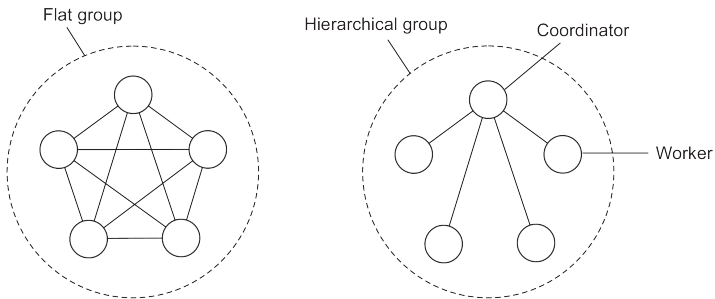
## Types of redundancy

- **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 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**).

# 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  $\mathbf{P} = \{P_1, \dots, P_n\}$
- **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

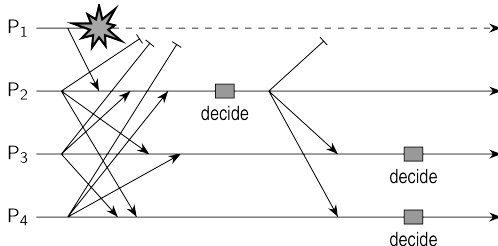
## System model

- A process group  $\mathbf{P} = \{P_1, \dots, P_n\}$
- **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  $\mathbf{C}_i^r$  to all others
2. At the end of  $r$ , each  $P_i$  merges all received commands into a new  $\mathbf{C}_i^{r+1}$ .
3. Next command  $cmd_i$  selected through a **globally shared, deterministic function**:  $cmd_i \leftarrow select(\mathbf{C}_i^{r+1})$ .

## 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  $P_4$ ).

# Raft

## Developed for understandability

- Uses a fairly straightforward **leader-election** algorithm (see Chp. 5). The current leader operates during the **current term**.
- Every server (typically, five) keeps a **log** of operations, some of which have been committed. **A backup will not vote for a new leader if its own log is more up to date.**
- All committed operations have the same position in the log of each respective server.
- The leader decides which pending operation is to be committed next  $\Rightarrow$  a **primary-backup approach**.

# Raft

## When submitting an operation

- A client submits a request for operation  $o$ .
- The leader appends the request  $\langle o, t, \rangle$  to its own log (registering the current term  $t$  and length of ).
- The log is (conceptually) broadcast to the other servers.
- The others (conceptually) copy the log and acknowledge the receipt.
- When a majority of acks arrives, the leader commits  $o$ .

# Raft

## When submitting an operation

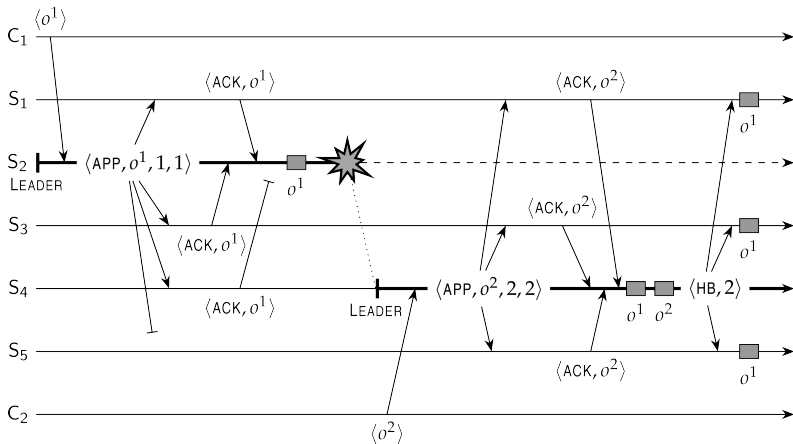
- A client submits a request for operation  $o$ .
- The leader appends the request  $\langle o, t, \rangle$  to its own log (registering the current term  $t$  and length of  $o$ ).
- The log is (conceptually) broadcast to the other servers.
- The others (conceptually) copy the log and acknowledge the receipt.
- When a majority of acks arrives, the leader commits  $o$ .

### Note

In practice, only updates are broadcast. At the end, every server has the same view and knows about the  $c$  committed operations. Note that effectively, any information at the backups is overwritten.



## Raft: when a leader crashes



### Crucial observations

- The new leader has the most committed operations in its log.
- Any missing commits will eventually be sent to the other backups.

# 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

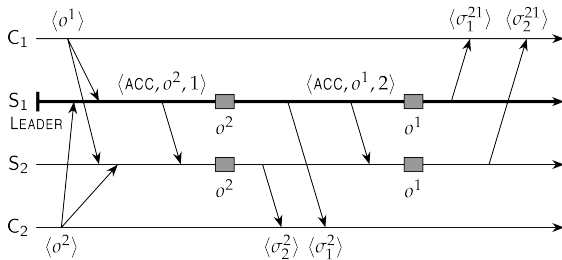
We will build up Paxos from scratch to understand where many consensus algorithms actually come from.

# 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)  $\Rightarrow$  we consider only **control messages**.

## Two-server situation

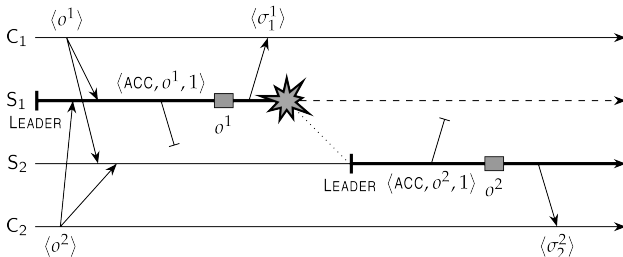


# Handling lost messages

## 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.

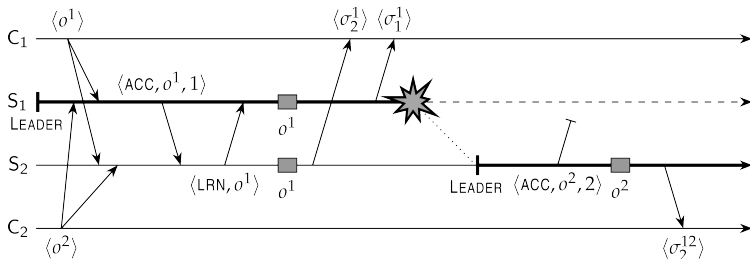
## Two servers and one crash: problem



### Problem

Primary crashes after executing an operation, but the backup never received the accept message.

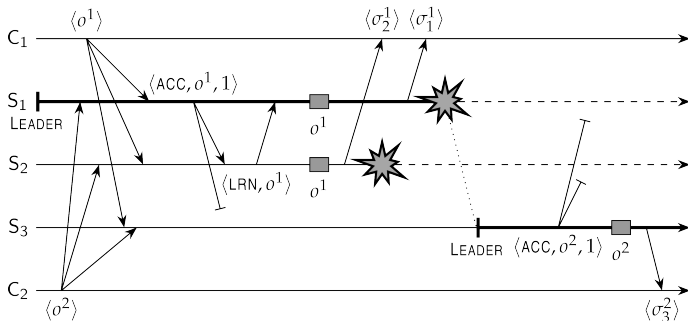
## Two servers and one crash: solution



### Solution

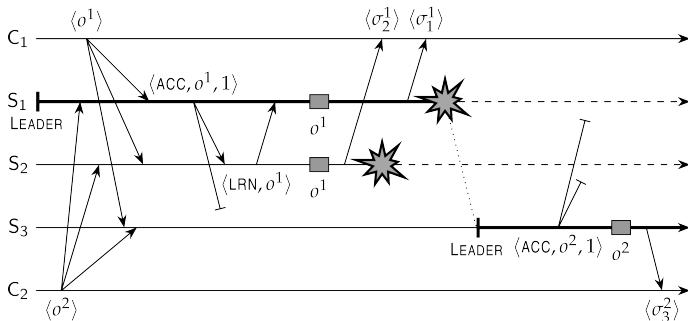
Never execute an operation before it is clear that it has been learned.

## Three servers and two crashes: still a problem?





## Three servers and two crashes: still a problem?



### Scenario

What happens when  $\text{LEARN}(o^1)$  as sent by  $S_2$  to  $S_1$  is lost?

[illegible]

## 26 / 77

## Paxos: fundamental rule

### General 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

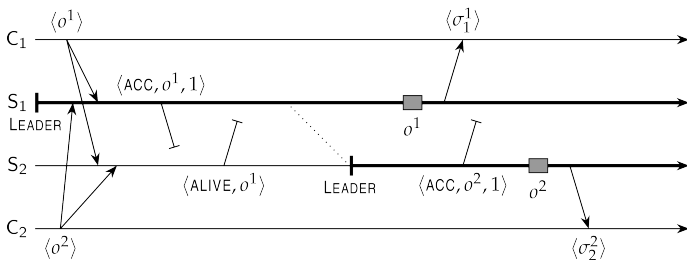
## Practice

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.



## Required number of servers

### 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  $\text{ACCEPT}(o, 1)$ .
- If  $S_2$  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  $\text{ACCEPT}(o, 1)$ .
- If  $S_2$  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
- If  $S_2$  sends  $\text{ACCEPT}(o^1, 1) \Rightarrow S_3$  retransmits  $\text{LEARN}(o^1)$ .
- If  $S_2$  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 $\text{ACCEPT}(o^1, 1)$

$S_3$  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  $o^1$ .

## 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.

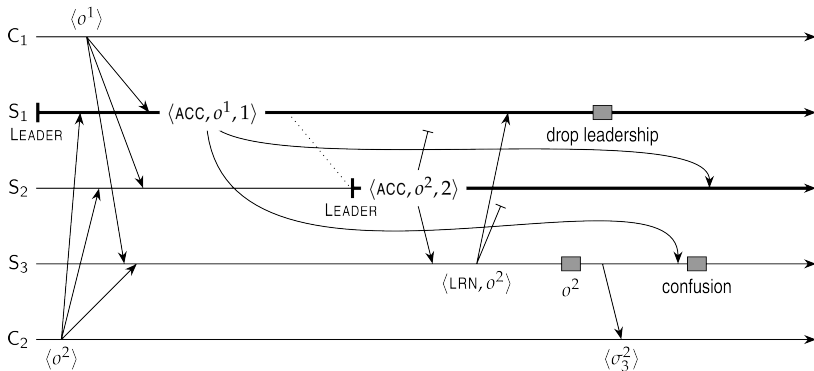
$S_2$  had missed  $\text{ACCEPT}(o^1, 1)$

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.

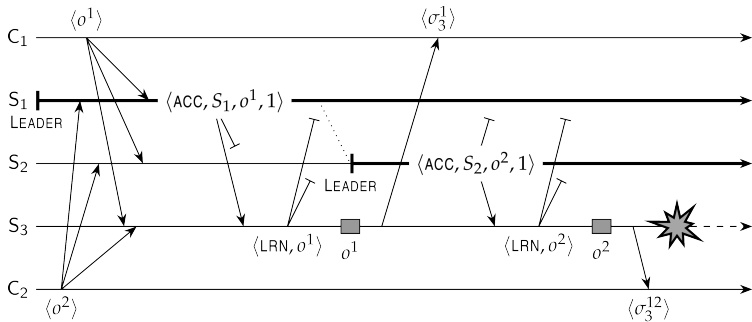
## False crash detections



### Problem and solution

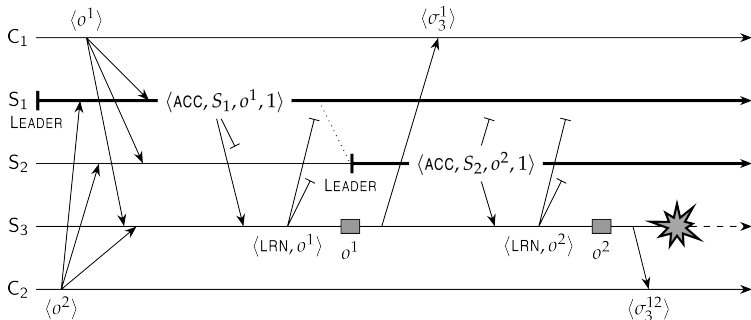
$S_3$  receives  $\text{ACCEPT}(o^1, 1)$ , but much later than  $\text{ACCEPT}(o^2, 1)$ . If it knew who the **current** leader was, it could safely reject the delayed accept message  $\Rightarrow$  leaders should include their ID in messages.

## But what about progress?





## But what about progress?



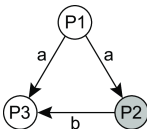
### Essence of solution

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.

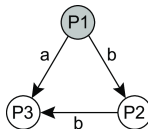
# Consensus under arbitrary failure semantics

## Essence

We consider process groups in which communication between process is **inconsistent**.



Improper forwarding



Different messages

# Consensus under arbitrary failure semantics

## System model

- We consider a **primary**  $P$  and  $n - 1$  **backups**  $B_1, \dots, B_{n-1}$ .
- 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

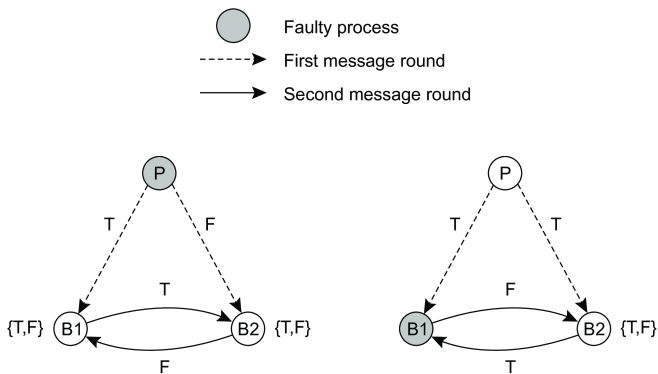
**BA1:** Every nonfaulty backup process stores the same value.

**BA2:** If the primary is nonfaulty then every nonfaulty backup process stores exactly what the primary had sent.

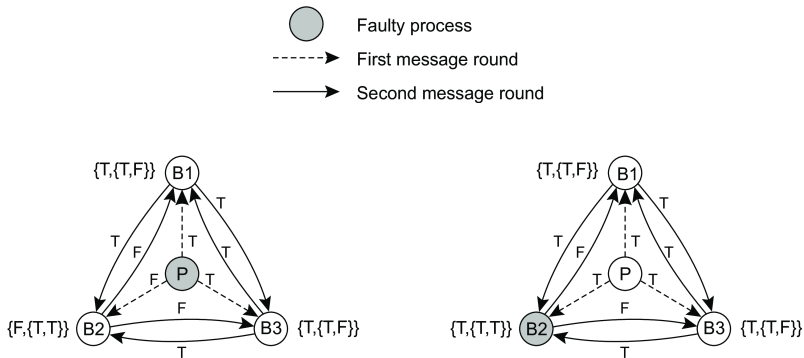
## Observation

- 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



# Practical Byzantine Fault Tolerance (PBFT)

## Background

One of the first solutions that managed to Byzantine fault tolerance while keeping performance acceptable. Popularity has increased with the introduction of [permissioned blockchains](#).

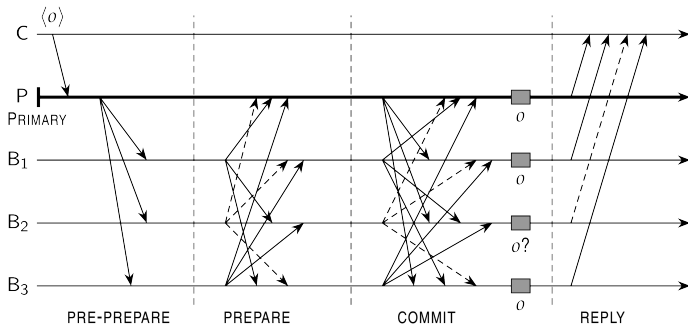
## Assumptions

- A server may exhibit arbitrary failures
- Messages may be lost, delayed, and received out of order
- Messages have an [identifiable sender](#) (i.e., they are [signed](#))
- [Partially synchronous](#) execution model

## Essence

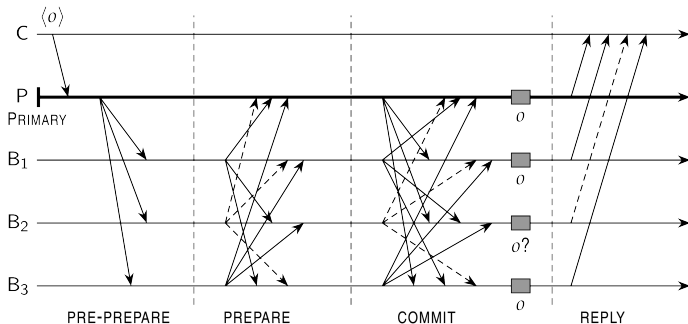
A [primary-backup approach](#) with  $3k + 1$  replica servers.

## PBFT: four phases



- $C$  is the client
- $P$  is the primary
- $B_1, B_2, B_3$  are backups
- Assume  $B_2$  is faulty

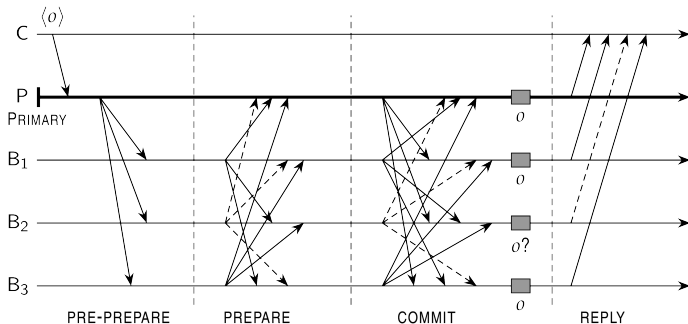
## PBFT: four phases



- All servers assume to be working in a current **view**  $v$ .
- $C$  requests operation  $o$  to be executed
- $P$  **timestamps**  $o$  and sends **PRE-PREPARE**( $t, v, o$ )
- Backup  $B_i$  accepts the pre-prepare message if it is also in  $v$  and has not accepted a an operation with timestamp  $t$  before.

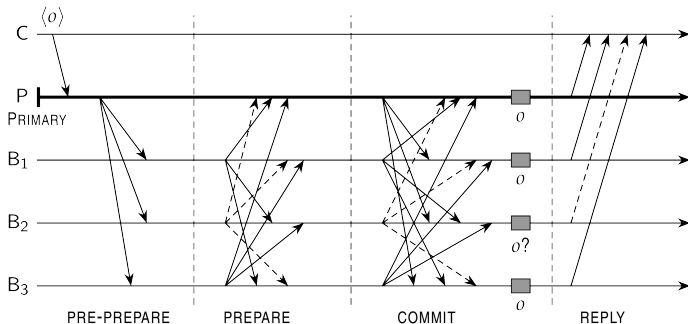


## PBFT: four phases



- $B_i$  broadcasts  $\text{PREPARE}(t, v, o)$  to all (including the primary)
- **Note:** a nonfaulty server will eventually log  $2k$  messages  $\text{PREPARE}(t, v, o)$  (including its own)  $\Rightarrow$  consensus on the ordering of  $o$ .
- **Note:** it doesn't matter what faulty  $B_2$  sends, it cannot affect joint decisions by  $P, B_1, B_3$ .

## PBFT: four phases



- All servers broadcast **COMMIT**( $t, v, o$ )
- The commit is needed to also make sure that  $o$  can be executed **now**, that is, in the current view  $v$ .
- When  $2k$  messages have been collected, excluding its own, the server can safely execute  $o$  en reply to the client.

# PBFT: when the primary fails

## Issue

When a backup detects the primary failed, it will broadcast a **view change** to view  $v + 1$ . We need to ensure that any **outstanding request** is executed **once and only once** by all nonfaulty servers. The operation needs to be handed over to the new view.

## Procedure

- The next primary  $P^*$  is known deterministically
- A backup server broadcasts **VIEW-CHANGE**( $v + 1, \mathbf{P}$ ):  $\mathbf{P}$  is the set of prepares it had sent out.
- $P^*$  waits for  $2k + 1$  view-change messages, with  $\mathbf{X} = \bigcup \mathbf{P}$  containing all previously sent prepares.
- $P^*$  sends out **NEW-VIEW**( $v+1, \mathbf{X}, \mathbf{O}$ ) with  $\mathbf{O}$  a new set of pre-prepare messages.
- **Essence**: this allows the nonfaulty backups to **replay** what has gone on in the previous view, if necessary, and bring  $o$  into the new view  $v + 1$ .

# 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

Process behavior	Message ordering				Commun. delay
	Unordered		Ordered		
	Unicast	Multicast	Unicast	Multicast	
Synchronous {	✓	✓	✓	✓	Bounded
			✓	✓	Unbounded
Asynchronous {				✓	Bounded
				✓	UnBounded
	Unicast	Multicast	Unicast	Multicast	
Message transmission					

### Formal requirements for consensus

- Processes produce the same output value
- Every output value must be valid
- Every process must eventually provide output

# 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.

# CAP theorem intuition

## Simple situation: two interacting processes

- $P$  and  $Q$  can no longer communicate:
  - Allow  $P$  and  $Q$  to go ahead  $\Rightarrow$  no consistency
  - Allow only one of  $P$ ,  $Q$  to go ahead  $\Rightarrow$  no availability
- $P$  and  $Q$  have to be assumed to continue communication  $\Rightarrow$  no partitioning allowed.

# CAP theorem intuition

## Simple situation: two interacting processes

- $P$  and  $Q$  can no longer communicate:
  - Allow  $P$  and  $Q$  to go ahead  $\Rightarrow$  no consistency
  - Allow only one of  $P$ ,  $Q$  to go ahead  $\Rightarrow$  no availability
- $P$  and  $Q$  have to be assumed to continue communication  $\Rightarrow$  no partitioning allowed.

## Fundamental question

What are the practical ramifications of the CAP theorem?



# 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

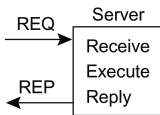
## What can go wrong?

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.

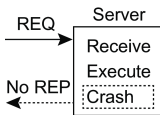
## Two “easy” solutions

- 1: (cannot locate server): just report back to client
- 2: (request was lost): just resend message

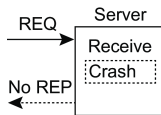
## Reliable RPC: server crash



(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.

# 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

# Why fully transparent server recovery is impossible

Reissue strategy	Strategy $M \rightarrow P$			Strategy $P \rightarrow M$		
	MPC	MC(P)	C(MP)	PMC	PC(M)	C(PM)
Always	DUP	OK	OK	DUP	DUP	OK
Never	OK	ZERO	ZERO	OK	OK	ZERO
Only when ACKed	DUP	OK	ZERO	DUP	OK	ZERO
Only when not ACKed	OK	ZERO	OK	OK	DUP	OK
<b>Client</b>	<b>Server</b>			<b>Server</b>		
	OK = Document processed once					
	DUP = Document processed twice					
	ZERO = Document not processed at all					

## 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).

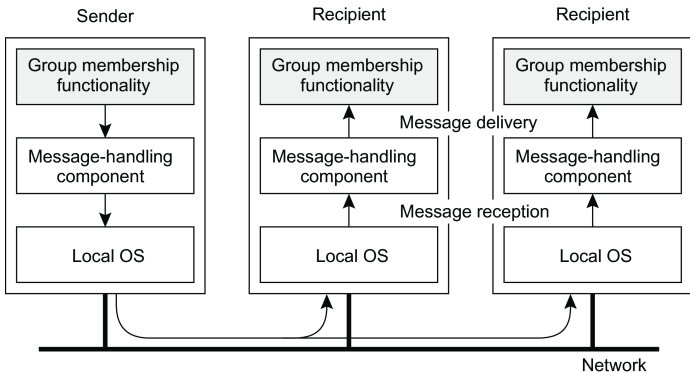
## Solution

- **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.

# 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.



## 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.

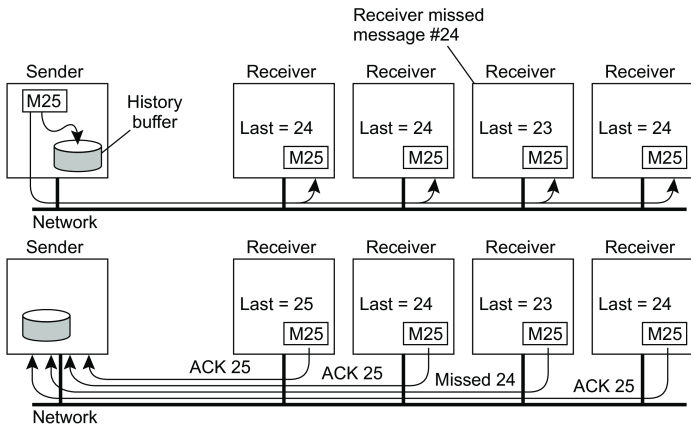
### Tricky part

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.

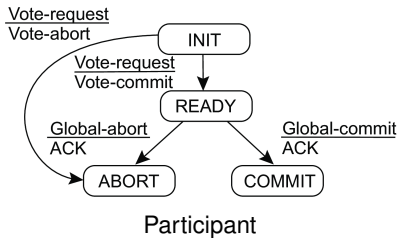
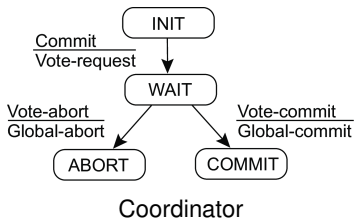
## Two-phase commit protocol (2PC)

### 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



## 2PC – Failing participant

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

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



## 2PC – Failing participant

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.

## 2PC – Failing participant

### Alternative

When a recovery is needed to *READY* state, check state of other participants  
⇒ no need to log coordinator's decision.

Recovering participant *P* contacts another participant *Q*

State of <i>Q</i>	Action by <i>P</i>
<i>COMMIT</i>	Make transition to <i>COMMIT</i>
<i>ABORT</i>	Make transition to <i>ABORT</i>
<i>INIT</i>	Make transition to <i>ABORT</i>
<i>READY</i>	Contact another participant

### 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.

## 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

```
1 class Coordinator:
2     def run(self):
3         yetToReceive = list(self.participants)
4         self.log.info('WAIT')
5         self.chan.sendTo(self.participants, VOTE_REQUEST)
6         while len(yetToReceive) > 0:
7             msg = self.chan.recvFrom(self.participants, BLOCK, TIMEOUT)
8             if msg == -1 or (msg[1] == VOTE_ABORT):
9                 self.log.info('ABORT')
10                self.chan.sendTo(self.participants, GLOBAL_ABORT)
11                return
12            else: # msg[1] == VOTE_COMMIT
13                yetToReceive.remove(msg[0])
14                self.log.info('COMMIT')
15                self.chan.sendTo(self.participants, GLOBAL_COMMIT)
```

# Participant in Python

```
1 class Participant:
2     def run(self):
3         self.log.info('INIT')
4         msg = self.chan.recvFrom(self.coordinator, BLOCK, TIMEOUT)
5         if msg == -1: # Crashed coordinator - give up entirely
6             decision = LOCAL_ABORT
7         else: # Coordinator will have sent VOTE_REQUEST
8             decision = self.do_work()
9             if decision == LOCAL_ABORT:
10                 self.chan.sendTo(self.coordinator, VOTE_ABORT)
11                 self.log.info('LOCAL_ABORT')
12             else: # Ready to commit, enter READY state
13                 self.log.info('READY')
14                 self.chan.sendTo(self.coordinator, VOTE_COMMIT)
15                 msg = self.chan.recvFrom(self.coordinator, BLOCK, TIMEOUT)
16                 if msg == -1: # Crashed coordinator - check the others
17                     self.log.info('NEED_DECISION')
18                     self.chan.sendTo(self.participants, NEED_DECISION)
19                 while True:
20                     msg = self.chan.recvFromAny()
21                     if msg[1] in [GLOBAL_COMMIT, GLOBAL_ABORT, LOCAL_ABORT]:
22                         decision = msg[1]
23                         break
24                 else: # Coordinator came to a decision
25                     decision = msg[1]
26             if decision == GLOBAL_COMMIT:
27                 self.log.info('COMMIT')
28             else: # decision in [GLOBAL_ABORT, LOCAL_ABORT]:
29                 self.log.info('ABORT')
30             while True: # Help any other participant when coordinator crashed
31                 msg = self.chan.recvFrom(self.participants)
32                 if msg[1] == NEED_DECISION:
33                     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

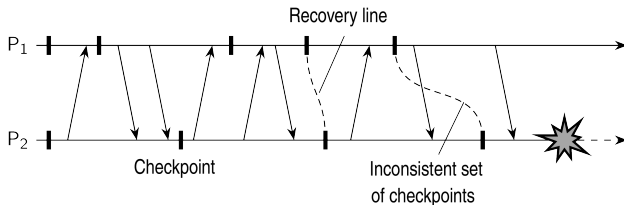
## Consistent recovery state

### 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

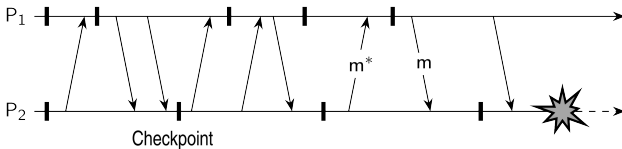
## Observation

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

# Independent checkpointing

## Essence

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

- Let  $CP_i(m)$  denote  $m^{\text{th}}$  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  $m^{\text{th}}$  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  $m^{\text{th}}$  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  $m^{\text{th}}$  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  $\Rightarrow$  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

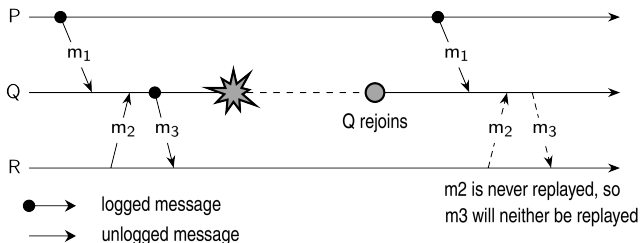
If we record nondeterministic events (to replay them later), we obtain a deterministic execution model that will allow us to do a complete replay.

## 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.





# Message-logging schemes

## Notations

- **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  $Q \in \mathbf{DEP}(m)$ .
- **COPY**( $m$ ): processes that have a copy of  $m$ , but have not (yet) reliably stored it.
- **FAIL**: the collection of crashed processes.

## Characterization

$Q$  is orphaned  $\Leftrightarrow \exists m : Q \in \mathbf{DEP}(m)$  and  $\mathbf{COPY}(m) \subseteq \mathbf{FAIL}$

# Message-logging schemes

## Pessimistic protocol

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

## 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 **COPY**( $m$ )  $\subseteq$  **FAIL**, then eventually also **DEP**( $m$ )  $\subseteq$  **FAIL**.

## Consequence

To guarantee that **DEP**( $m$ )  $\subseteq$  **FAIL**, we generally roll back each orphan process  $Q$  until  $Q \notin$  **DEP**( $m$ ).