JC4001: Distributed Systems

Topic 5: Fault Tolerance

Xiaonan Liu
xiaonan.liu@abdn.ac.uk

Content

Content

- Distributed Commit
- Recovery
  - Introduction
  - Checkingpoint
  - Message Logging

Introduction

- Basic Concepts

Basics

A component is any part of the system that provides services to clients. To provide these services, a component might need to use services from other components. This means that a component can depend on other components. For example, in a web application. The front-end (what users interact with) depends on the back-end (the server and database). If the back-end fails, the front-end won't work properly.

Specifically

A component C depends on another component C$*$ if the correctness of C's behavior depends on the correctness of C$*$'s behavior. In other words, if C$*$ fails, C won't work right either. Here, components can be processes (running programs) or channels (communication links between processes). So, understanding these dependencies is important for designing reliable and fault-tolerant systems. If you know which components depend on which, you can better manage failures and ensure the overall system keeps running smoothly.

Introduction

- Basic Concepts

  Fault-tolerant systems are part of what we call dependable systems

  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

- Availability: It means the system is ready to be used at any moment. It's about the probability that the system is working correctly and is available to perform its functions for users. In simple terms, a highly available system is one that's almost always up and running when you need it.

Introduction

- Basic Concepts

  Fault-tolerant systems are part of what we call dependable systems

  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: It means the system can run continuously without failing. Unlike availability, reliability is measured over a period of time. So, a highly reliable system is one that can keep working without interruptions for a long stretch of time. For example, if a system goes down for a millisecond every hour, it has high availability (over 99.9999%) but is still unreliable. However, a system that never crashes but shuts down for two weeks every August is highly reliable but has only 96% availability. See the difference?

Introduction

- Basic Concepts

  Fault-tolerant systems are part of what we call dependable systems

  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

- Safety: It is about ensuring that even if the system fails temporarily, nothing catastrophic happens. This is important for systems controlling things like nuclear power plants or space missions. A brief failure in such systems could lead to disastrous consequences. Building truly safe systems is incredibly challenging.

Introduction

- Basic Concepts

  Fault-tolerant systems are part of what we call dependable systems

  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

- Maintainability: It refers to how easily a failed system can be repaired. A system that's easy to fix can often maintain high availability, especially if it can detect and repair failures automatically. However, as we'll see, automatic recovery from failures is easier said than done.

Introduction

- Basic Concepts

Availability $A(t)$ of component C

Availability is all about how often a component is working correctly over a certain time period. Formally, we define the availability A(t) of a component during the time interval from 0 to t as the average fraction of time that the component has been working properly during that interval. If we think about long-term availability, we look at A(∞), which means how available the component is over an infinite period.

Reliability $R(t)$ of component C

It's about the likelihood that a component continues to work correctly throughout a certain time interval. Traditionally, we define the reliability R(t) of a component during the time interval from 0 to t as the probability that it has been working correctly during that time, assuming it was working fine at the start (time T=0).

While availability looks at how often a component is up and running, reliability focuses on the continuous functioning of the component over a period of time. Both are important for understanding how dependable a system is.

Introduction

- Basic Concepts

Traditional metrics

- Mean Time To Failure (*MTTF*): The average time a component works before it fails. Think of it as the expected lifespan of the component
- Mean Time To Repair (*MTTR*): The average time it takes to fix a component once it has failed. It gives us an idea of how quickly we can get things back up and running.
- Mean Time Between Failures (*MTBF*): Simply *MTTF* + *MTTR*. It represents the average time between one failure and the next.

Introduction

- Basic Concepts

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.

Introduction

- Basic Concepts

Security

A system is said to fail when it doesn’t reach its goals. For example, if a distributed system is supposed to provide several services to its users, it has failed if one or more of those services can't be fully delivered. An error is a part of the system’s state that could lead to a failure. For example, when sending packets over a network, it’s normal for some packets to get damaged. Damaged packets mean the receiver might read the wrong bit values, like seeing a 1 instead of a 0, or might not detect the packet at all. The cause of an error is called a fault. a faulty transmission medium could damage packets, and replacing the medium is a straightforward fix. However, if the error is due to bad weather in a wireless network, fixing the weather is a bit more challenging! Consider another example: a program crash. This is a failure, and it might happen because the program hit a piece of buggy code. The bug is the error, and the fault is the programmer who introduced it. So, the programmer is indirectly the cause of the crash.

Introduction

- Basic Concepts

Security

Building dependable systems is all about controlling faults. We can prevent, tolerate, remove, or forecast faults. The key is fault tolerance—making sure the system continues to work even when there are faults. For example, using error-correcting codes in packet transmission helps the system tolerate poor transmission lines, reducing the chance that a damaged packet leads to a failure. Faults are generally classified into three types: transient, intermittent, and permanent.

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

Introduction

- Basic Concepts

Faults

- Transient faults: happen once and then disappear. If you try the operation again, it usually works. Think of a bird flying through a microwave transmission beam, causing some lost bits. Retry the transmission, and it'll probably work the next time.
- Intermittent faults: come and go on their own. A loose connector often causes these kinds of faults. They're frustrating because they're hard to diagnose—everything seems fine when the repair person shows up!
- Permanent faults: continues to exist until the faulty component is replaced. Examples include burnt-out chips, software bugs, and disk-head crashes.

Introduction

- Basic Concepts

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

Introduction

- Basic Concepts

  Faults

  - Fault Prevention: This involves techniques to prevent faults from happening in the first place. Understand it as building a system with high-quality components and rigorous testing to catch issues before they cause problems.
  - Fault Tolerance: This strategy focuses on ensuring the system can still function correctly even when faults occur. This is like having backup systems or using error-correcting codes to keep things running smoothly.
  - Fault Removal: This is about identifying and fixing faults that are already in the system. Regular maintenance and updates help to catch and correct issues before they cause failures.
  - Fault Forecasting: This involves predicting where faults are likely to occur and taking steps to address them before they cause problems. It's like using data analysis to spot trends and anticipate issues, so you can be proactive rather than reactive.

Introduction

- Failure Models

  - Basically, a system fails when it can't provide the services it's supposed to. this means that either the servers, the communication channels, or both aren't doing their jobs properly. But sometimes, the issue isn't just with the server that's failing. If that server relies on other servers to do its job, the real problem might be elsewhere.
  - These dependency relationships are common in distributed systems. For example, if a disk fails, it can cause problems for a file server that needs that disk to provide a highly available file system. If this file server is part of a bigger distributed database, the whole database might be affected because part of its data is now inaccessible.

Introduction

- Failure Models

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

Introduction

- Basic Concepts

  Faults

  - Crash Failure: This happens when a server stops working completely and doesn't communicate anymore. A classic example is an operating system crash where the only solution is to reboot. This is common with personal computers, which is why the reset button is often on the front of the case.
  - Omission Failure: This occurs when a server doesn't respond to a request. There are two types:
  - (1) Receive-Omission Failure: The server never gets the request, possibly because there was no thread listening for incoming requests.
  - (2) Send-Omission Failure: The server does the work but fails to send a response, possibly due to an overflowed send buffer. In this case, the server might have to be ready for the client to reissue the request.

Introduction

- Basic Concepts

  Failure

  - Timing Failure: This occurs when the server's response is too early or too late. For example, in streaming video, data arriving too soon or too late can cause problems. A common type of timing failure is a performance failure, where the response is too late.
  - Response Failure: This happens when the server's response is incorrect. There are two kinds:
  - (1) Value Failure: The server gives the wrong answer to a request. For example, a search engine that returns irrelevant results has a value failure.
  - (2) State-Transition Failure: The server reacts unexpectedly to a request. For instance, if the server gets a message it doesn't recognize and takes an inappropriate action, that's a state-transition failure.
  - Arbitrary (Byzantine) Failure: This is the most serious type of failure. In this case, the server produces incorrect output that can't be detected as wrong. These failures are unpredictable and can cause major issues.

Introduction

- Failure Models

Omission versus commission

Arbitrary failures are often related to malicious activities by processes. But it's not always clear if a failure is malicious or just a mistake. For example, if a computer with a poorly designed operating system negatively impacts other computers, is it acting maliciously or just poorly engineered? Because of this uncertainty, it's better to use a distinction that doesn't involve making a judgment:

- Omission failures: This happens when a component fails to do something it should have done. For example, a server not sending a response when it should.
- Commission failure: This occurs when a component does something it shouldn't have done. For example, a server sending an incorrect response.

Observation

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

Introduction

- Failure Models

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.

Introduction

- Failure Models

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

Introduction

- Failure Models

Halting type

- Fail-Stop Failures: These are crash failures that can be reliably detected. This might happen if we assume the communication links aren't faulty and process P can set a maximum delay for responses from Q. If Q doesn't respond within that time, P knows Q has crashed.
- Fail-Noisy Failures: These are similar to fail-stop failures, but P will only eventually figure out that Q has crashed. There might be an unknown period during which P's detections are unreliable.
- Fail-Silent Failures: We assume the communication links are fine, but P can't tell if Q has crashed or if it's just an omission failure (like missing a message).
- Fail-Safe Failures: These involve arbitrary failures by Q, but they are harmless. They don't cause any damage.
- Fail-Arbitrary Failures: Q can fail in any possible way, including ways that aren't detectable and are harmful to the correct behavior of other processes. This is the worst type of failure.

Introduction

- Failure Masking by Redundancy

To make a system fault-tolerant, the goal is to hide failures from other processes as much as possible. The key technique for masking faults is redundancy.

Types of redundancy

- Information redundancy: This involves adding extra bits of data to help recover from errors. For example, adding a Hamming code to transmitted data allows recovery from noise on the transmission line.
- Time redundancy: This means performing an action, and if necessary, doing it again. For example, in transactions, if a transaction fails, it can be retried without any negative effects because nothing has been finalized. Another example is retransmitting a request to a server if a response is not received. Time redundancy is especially useful for transient or intermittent faults.
- Physical redundancy: This involves adding extra hardware or processes to the system to tolerate the loss or malfunction of some components. For example, adding extra processes so if a few crash, the system still functions correctly. This type of redundancy can be implemented in hardware or software, and by replicating processes, we can achieve a high degree of fault tolerance.

Process Resilience

- Resilience by Process Groups

Basic idea

Protect against malfunctioning processes through process replication, organizing multiple processes into a process group. The main strategy for handling a faulty process is to organize several identical processes into a group. The key feature of these groups is that when a message is sent to the group, all members receive it. This way, if one process fails, another can take over. Process groups can be dynamic: new groups can be created, old groups can be destroyed, and processes can join or leave groups during operation. A process can even be part of multiple groups at once, so we need mechanisms to manage these groups and their memberships. The purpose of groups is to allow a process to interact with a collection of other processes as if they were a single entity. For example, process P can send a message to group Q={Q1,Q2,...,QN} without needing to know who or where they are, or how many there are. To P, the group Q looks like a single logical process.

Process Resilience

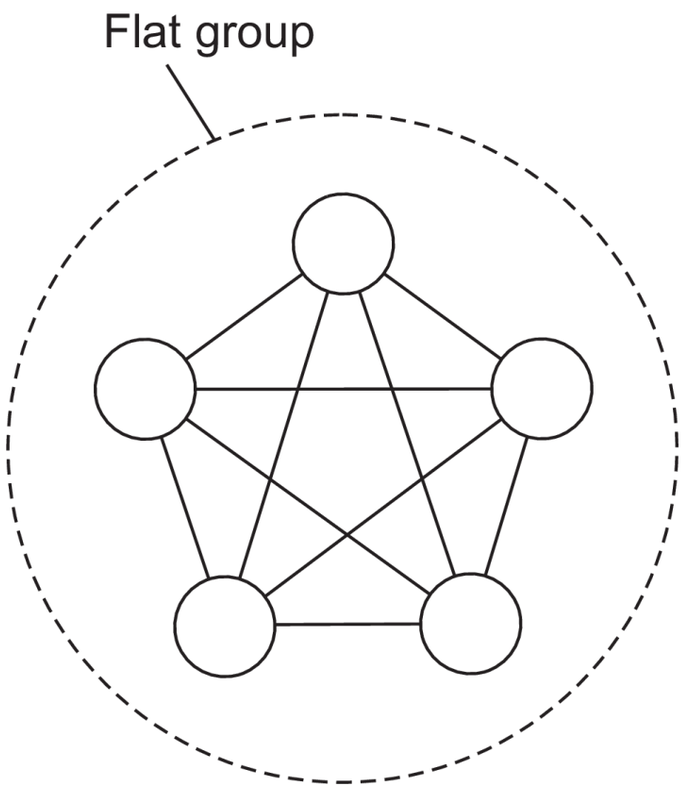
- Resilience by Process Groups

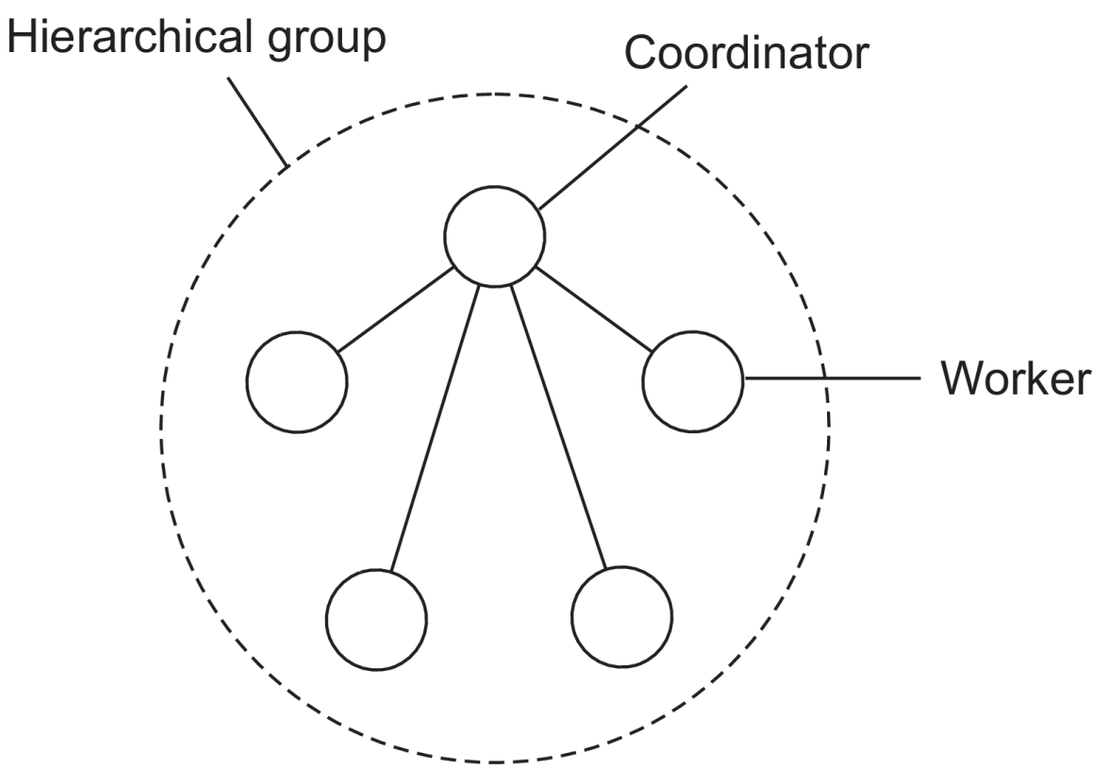
Basic idea

Distinguish between flat groups and hierarchical groups.

Flat Group

All processes are equal, forming a flat group with no leader. Decisions are made collectively, and if one process crashes, the group just gets smaller but keeps going. The downside is that decision-making can be slower because it often requires a vote.



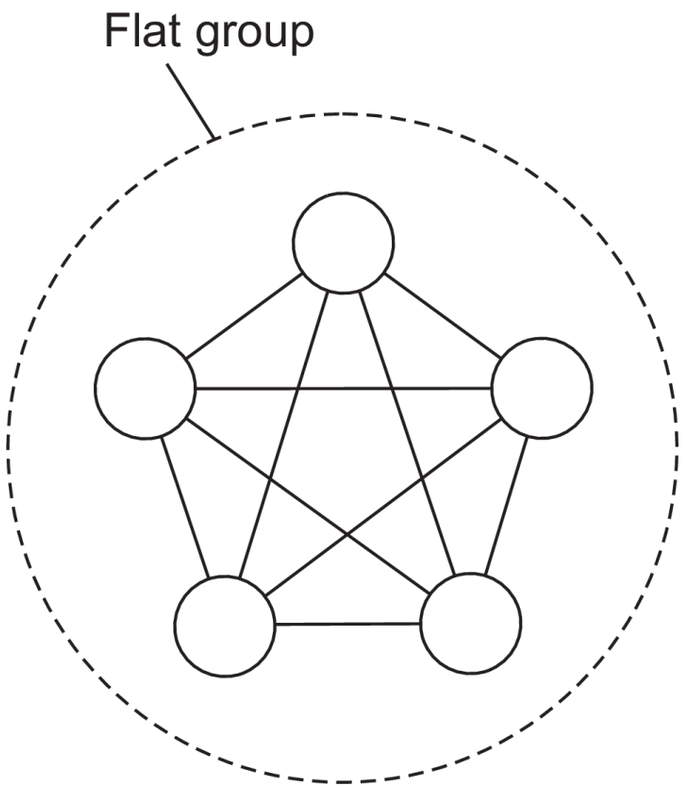


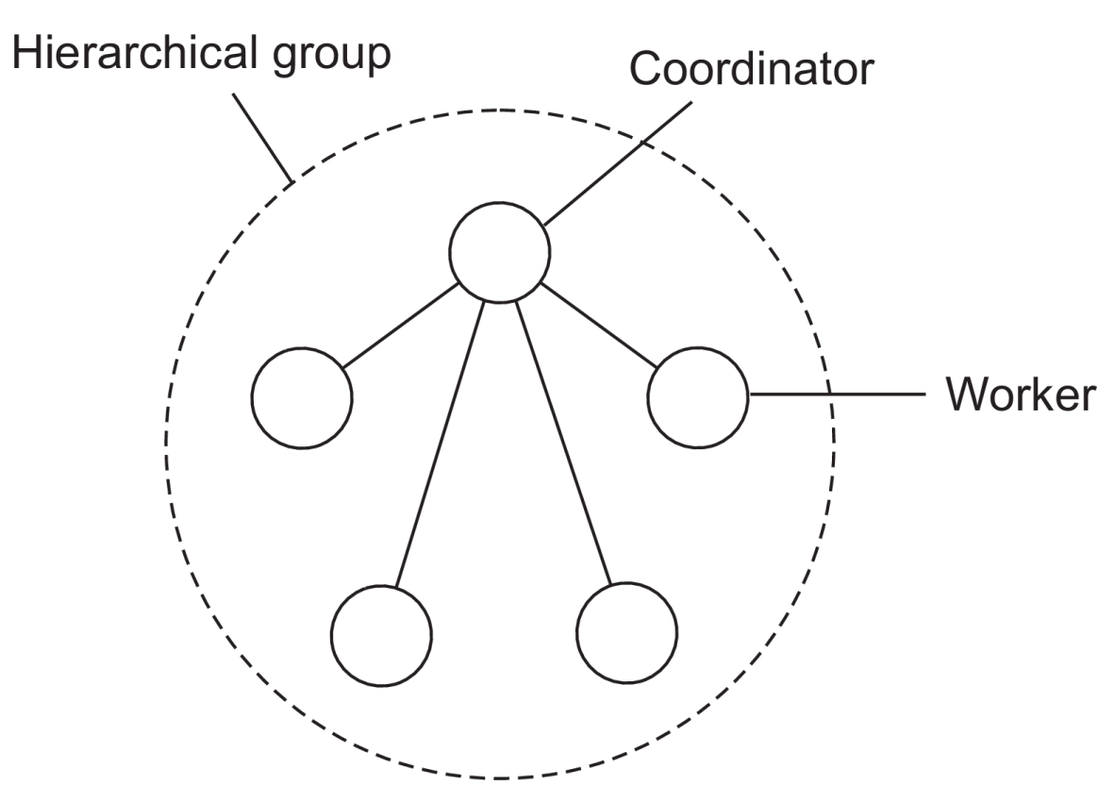
Process Resilience

- Resilience by Process Groups

Hierarchical Group

In hierarchical groups, there is a coordinator (or leader) and several workers. The coordinator handles requests and delegates tasks to the workers. This makes decision-making faster, but if the coordinator fails, the whole group stops until a new coordinator is elected. Examples of hierarchical groups include the Domain Name System and primary-backup schemes.





Process Resilience

- Resilience by Process Groups

  - Managing Group Membership: To create and manage groups, one approach is to use a group server that handles all requests for joining or leaving groups. This server keeps a database of all groups and their members. This method is simple and efficient but has the downside of being a single point of failure. If the group server crashes, group management stops, and all groups might need to be rebuilt.
  - Another approach is distributed group management, where members manage the group collectively. For instance, a process can broadcast a message to join a group. To leave a group, a process sends a goodbye message to everyone. However, detecting when a process crashes is trickier since there's no announcement—other members have to figure it out when the process stops responding.
  - Synchronization: When a process joins or leaves a group, this action must be synchronized with ongoing data messages. A new member must start receiving all group messages from the moment it joins, and a leaving member must stop receiving messages immediately. This can be managed by treating join or leave actions as messages sent to the whole group.

Process Resilience

- Resilience by Process Groups

  - Handling Multiple Failures: If many processes fail, the group might stop functioning. We need a protocol to rebuild the group, which often involves some process taking the lead. The protocol must handle situations where multiple processes try to take the lead simultaneously, possibly using a leader-election algorithm.

Process Resilience

- Failure Masking and Replication

k-fault tolerant group

When a group can mask any *k* concurrent member failures (*k* is called degree of fault tolerance). A system is k-fault tolerant if it can handle faults in k components and still function correctly.

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

- With halting failures (crash/omission/timing failures): we need a total of *k* + 1 members as no member will produce an incorrect result, so the result of one member is good enough. If k of them stop, the response from the remaining one can be used.
- With arbitrary failures: we need 2*k* + 1 members so that the correct result can be obtained through a majority vote. In the worst case, the k failing processes could all generate the same wrong reply, but the remaining k + 1 will produce the correct answer, so the client can trust the majority.

Important assumptions

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

Process Resilience

- Consensus in Faulty Systems with Crash Failures

Prerequisite

When it comes to clients and servers, we're using a model where a large group of clients send commands to a group of processes that work together like a single, super-reliable process. For this to work, there's an important assumption we need to make:

In a fault-tolerant process group, each non-faulty process must execute the same commands, in the same order, as every other non-faulty process.

Reformulation

Nonfaulty group members need to reach consensus on which command to execute next. If there were no failures, reaching consensus would be easy. For example, we could use Lamport's totally ordered multicasting or a centralized sequencer that hands out sequence numbers to each command. But since failures do happen, reaching consensus among a group of processes becomes much trickier in real-world scenarios.

Process Resilience

- Consensus in Faulty Systems with Crash Failures: Flooding-based Consensus

System model

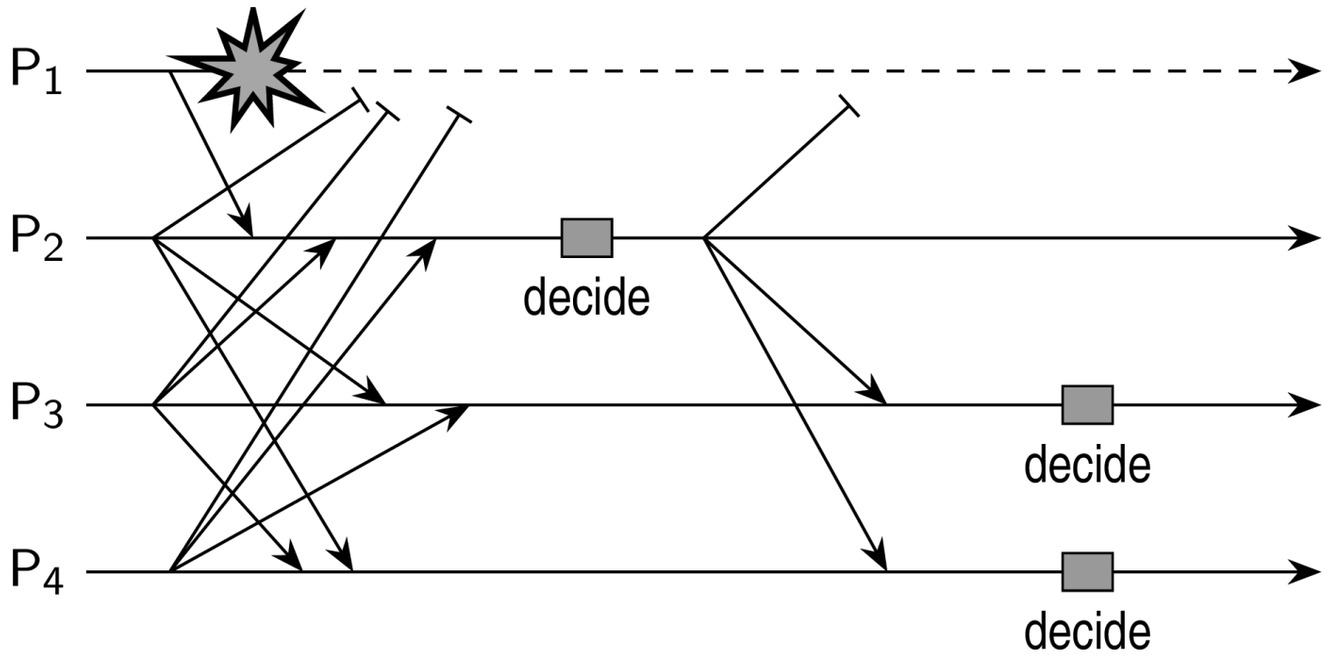
- A process group **P** = $\{P_1,\ldots,P_n\}$
- Work under fail-stop failure semantics, i.e., with reliable failure detection, this means we can reliably detect when one of the processes crashes
- A client contacts a P_i requesting it to execute a command
- Every P_i maintains a list of proposed commands

Basic algorithm (based on rounds)

1. In each round, a process Pi sends its list of proposed commands to every other process in the group.
2. At the end of the round, each process merges all the received proposed commands into a new list.
3. From this new list, each process deterministically selects the command to execute, if possible.

Process Resilience

- Consensus in Faulty Systems with Crash Failures: Flooding-based Consensus

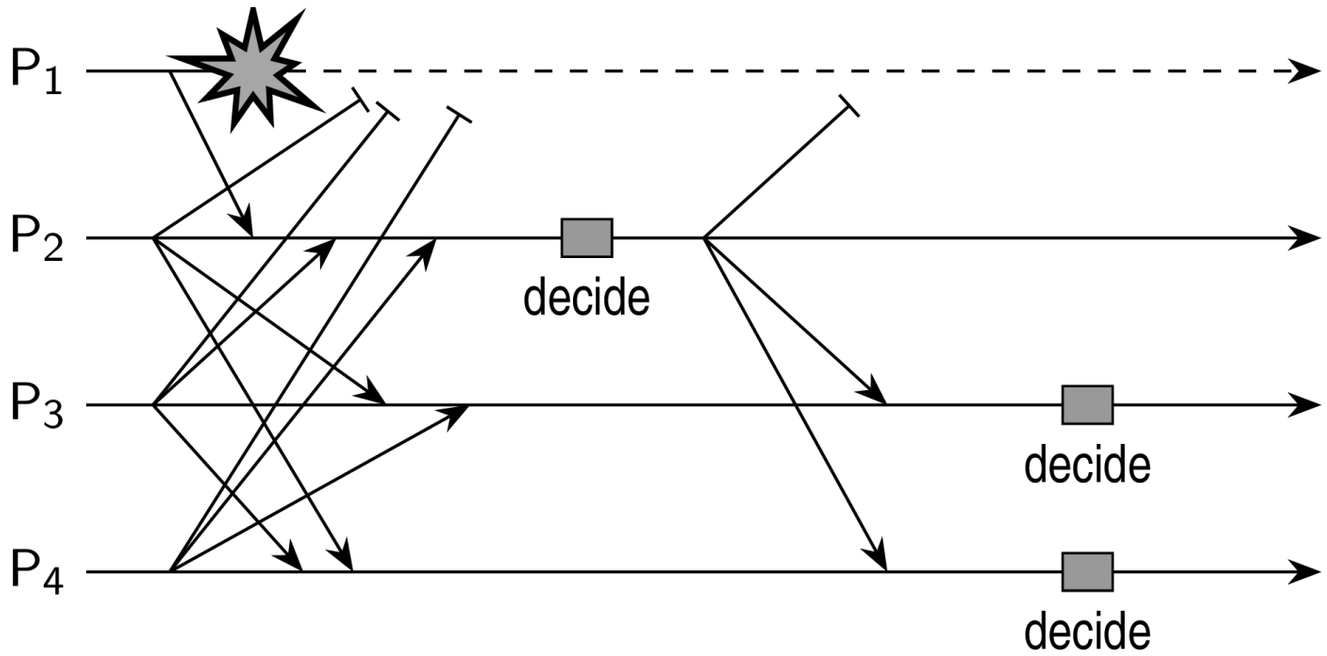


Observations

- This approach works well as long as none of the processes fail. But problems start when a process, say Pi, detects during round r that another process Pk has crashed. Suppose we have a group of four processes: P1,P2,P3, and P4. If P1 crashes during round r, and P2 received the list of proposed commands from P1 before it crashed, but P3 and P4 did not, we run into issues.

Process Resilience

- Consensus in Faulty Systems with Crash Failures: Flooding-based Consensus

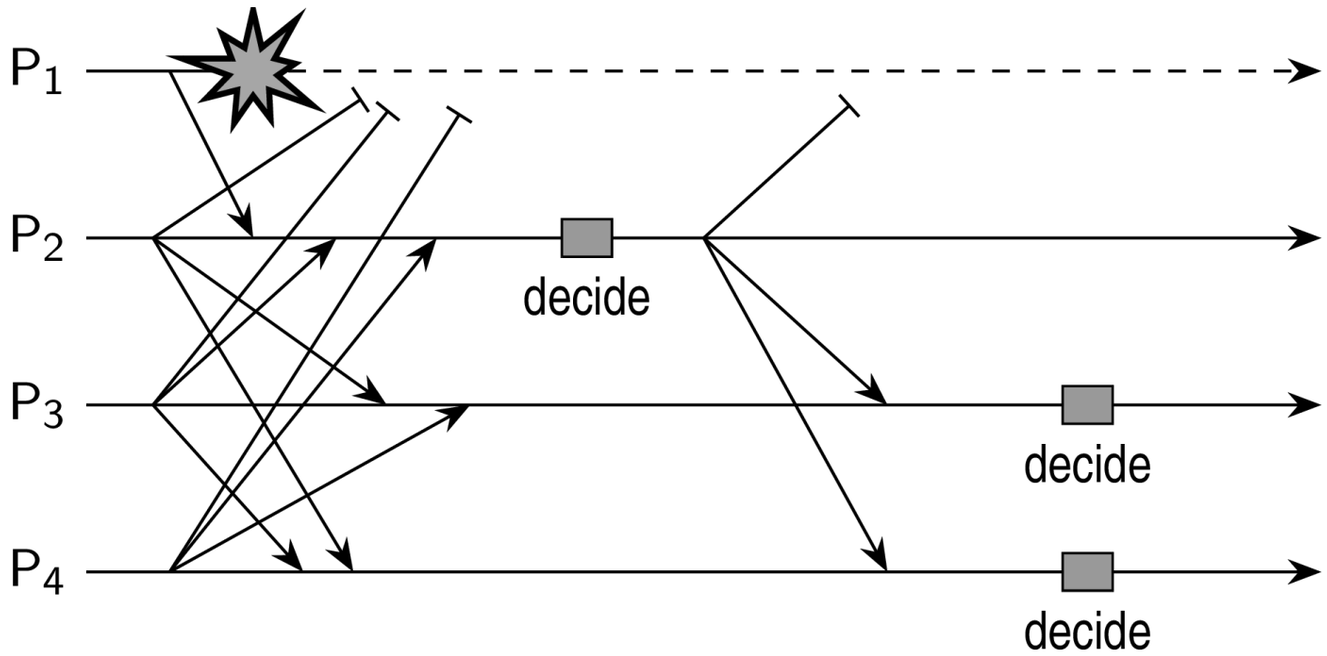


Observations

- P_2 received all proposed commands from all other processes ⇒ 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 ⇒ cannot make decision (same for P_4).

Process Resilience

- Consensus in Faulty Systems with Crash Failures: Flooding-based Consensus

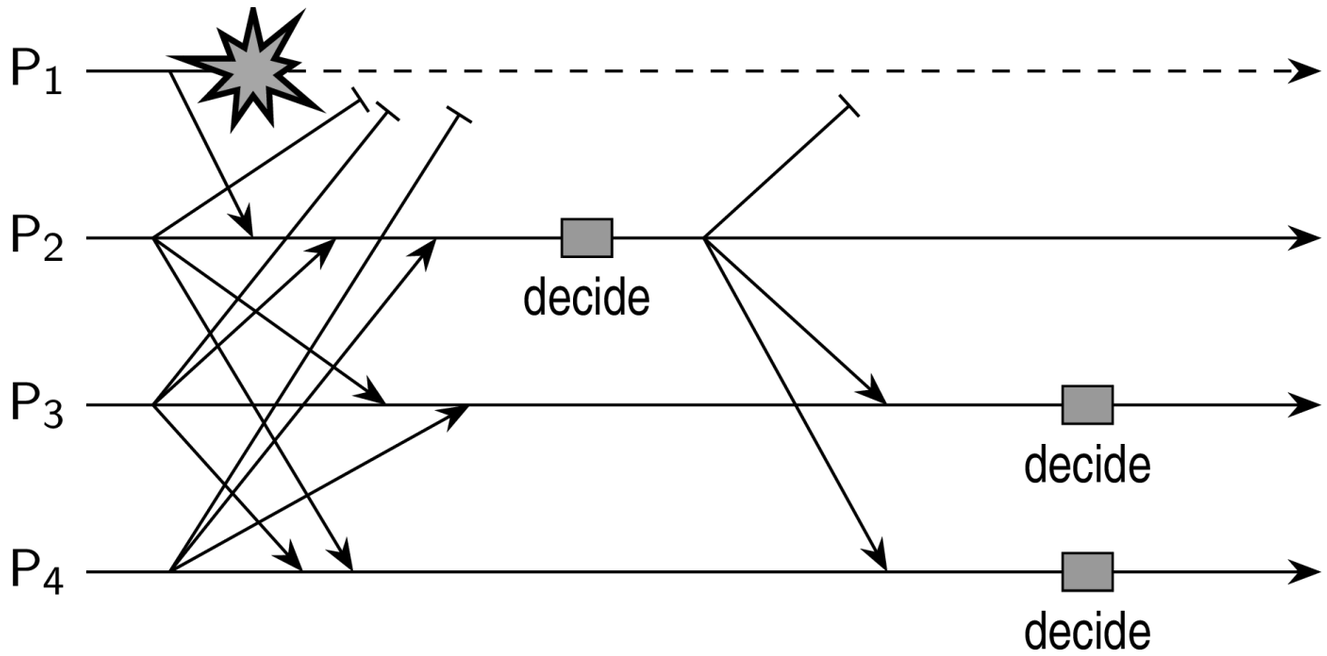


Observations

- If another process did get P1's commands, they might make a different decision than P3. So, the best P3 can do is wait until the next round to decide. The same goes for P4. A process will move to the next round only after receiving a message from every non-faulty process. This relies on reliable failure detection to know which processes are still active.

Process Resilience

- Consensus in Faulty Systems with Crash Failures: Flooding-based Consensus

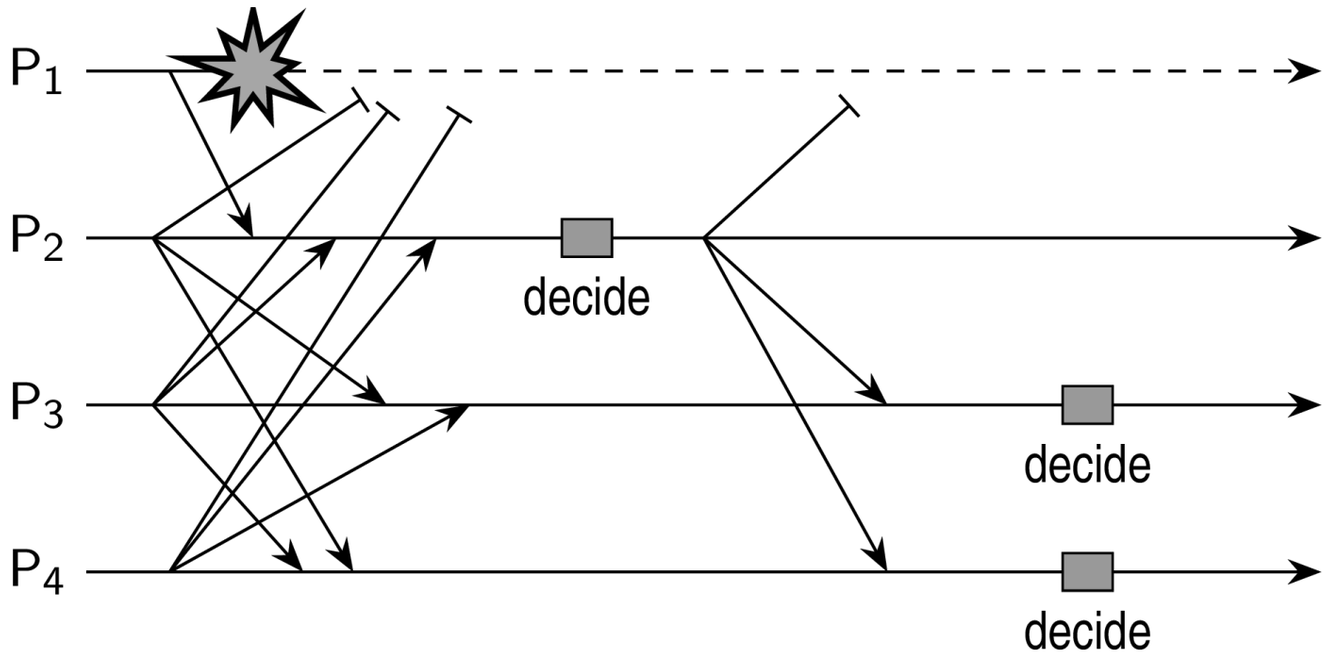


Observations

- Because P2 got all the commands, it can decide and broadcast that decision to the others. In the next round r+1, P3 and P4 can decide to execute the same command as P2. This algorithm works because a process only moves to the next round without deciding if it detects another process has failed. In the worst case, if only one non-faulty process is left, it can decide which command to execute.

Process Resilience

- Consensus in Faulty Systems with Crash Failures: Flooding-based Consensus



Observations

- This assumes we can reliably detect failures. But what if P2's decision message to P3 gets lost? P3 still can’t decide. Worse, P3 must make the same decision as P2 and P4. If P2 hasn’t crashed, it can retransmit its decision. If P2 did crash, P4 would detect this and rebroadcast its decision. P3 moves to the next round and, after receiving P4’s decision, completes the algorithm.

Process Resilience

- Consensus in Faulty Systems with Crash Failures: Flooding-based Consensus

  Developed for understandability

  - Uses a fairly straightforward leader-election algorithm. 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 ⇒ a primary-backup approach.
  - This setup ensures that the system is not only fault-tolerant but also straightforward to manage and understand.

Process Resilience

- Consensus in Faulty Systems with Crash Failures: Raft

Raft is a consensus protocol that works under crash failure, where a process will eventually figure out if another process has crashed.

In Raft, we typically have about five replicated servers. While Raft allows servers to join or leave the group, let's assume the set of servers is fixed. Each server keeps a log of operations, some of which are completed (committed) and some are pending. The goal is to reach consensus on these logs, meaning all committed operations should be in the same position in each server's log. One server acts as the leader, deciding the order of pending operations to be committed.

A client submits a request for operation *o*.

- Client Requests: A client sends an operation request to the leader, which keeps track of all pending requests. The leader appends each request to its log with a tuple ⟨operation, term, index⟩, where the term is the current term the leader is serving, and the index is the position in the log.

Process Resilience

- Consensus in Faulty Systems with Crash Failures: Raft

When submitting an operation

- Log Replication: The leader then sends its entire log (conceptually) to the other servers, along with the current index of the most recently committed operation. Each follower server copies the log and sends back an acknowledgment after ensuring all operations up to the current index are executed.
- Commitment: Once the leader gets a majority of acknowledgments, it commits the operation, informs the client, and updates the index of the committed operation. This index is then sent to the followers, so they can also commit the operation.
- Leader Crash: If the leader crashes, a new leader is elected. This new leader's log becomes the collective state of the server group. However, during the election, a server won't vote for a candidate if its own log is more up-to-date. A log is considered more up-to-date if it has more committed operations or contains operations from more recent terms.

Process Resilience

- Consensus in Faulty Systems with Crash Failures: Raft

When submitting an operation

- Consistency and Redundancy: If the new leader has executed all committed operations, its state will match that of the previous leader before it crashed. If a client doesn't get a response, it will resend its request, ensuring no requests are missed.

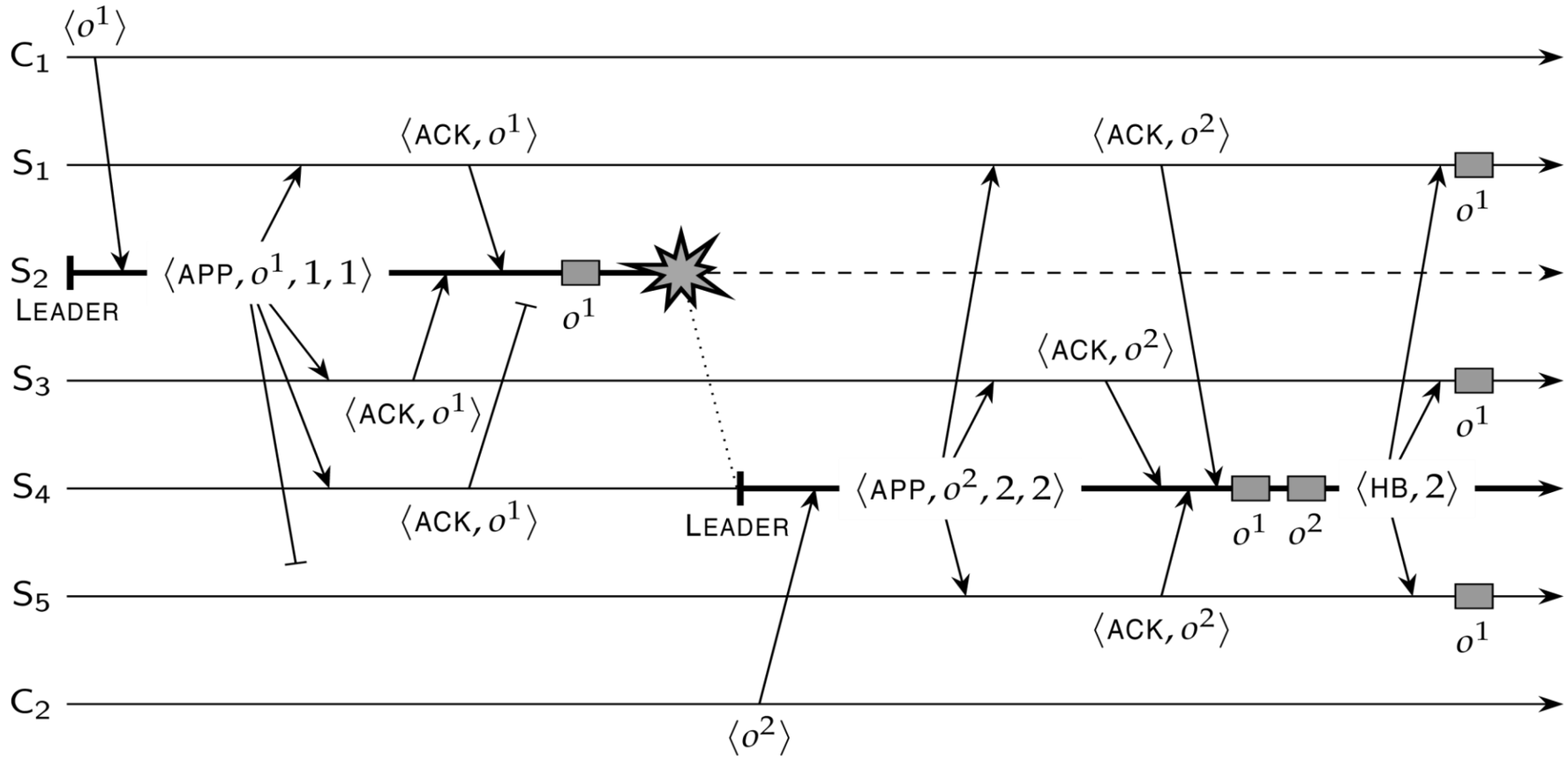
Note

Raft ensures that the leader's state is the collective state of the group, making sure all servers are on the same page and any crashes are handled smoothly.

Process Resilience

- Consensus in Faulty Systems with Crash Failures: Raft

What happens if the leader crashes after executing operation o1 but before it had a chance to inform all the other servers?

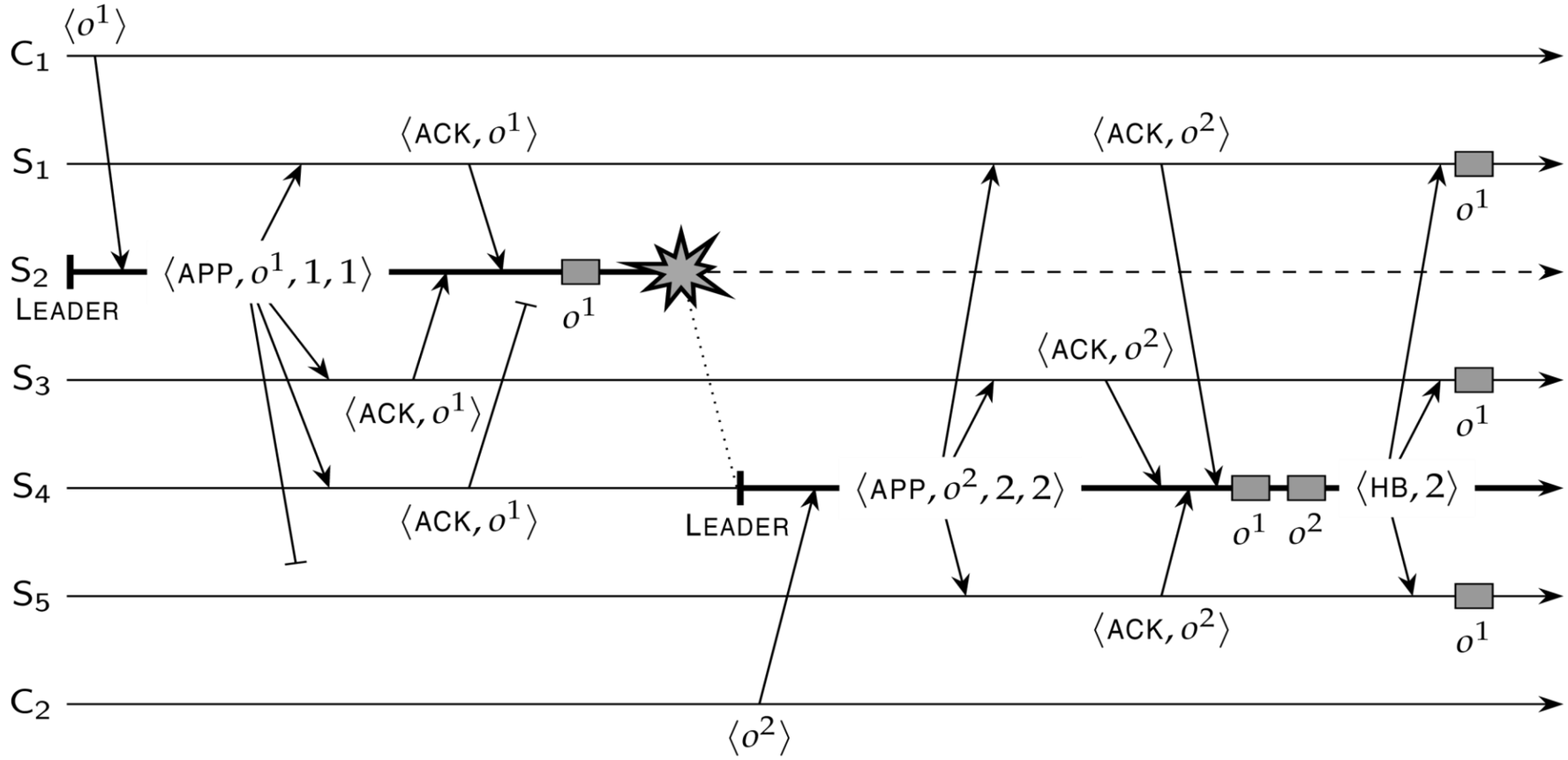


Crucial observations

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

Process Resilience

- Consensus in Faulty Systems with Crash Failures: Raft

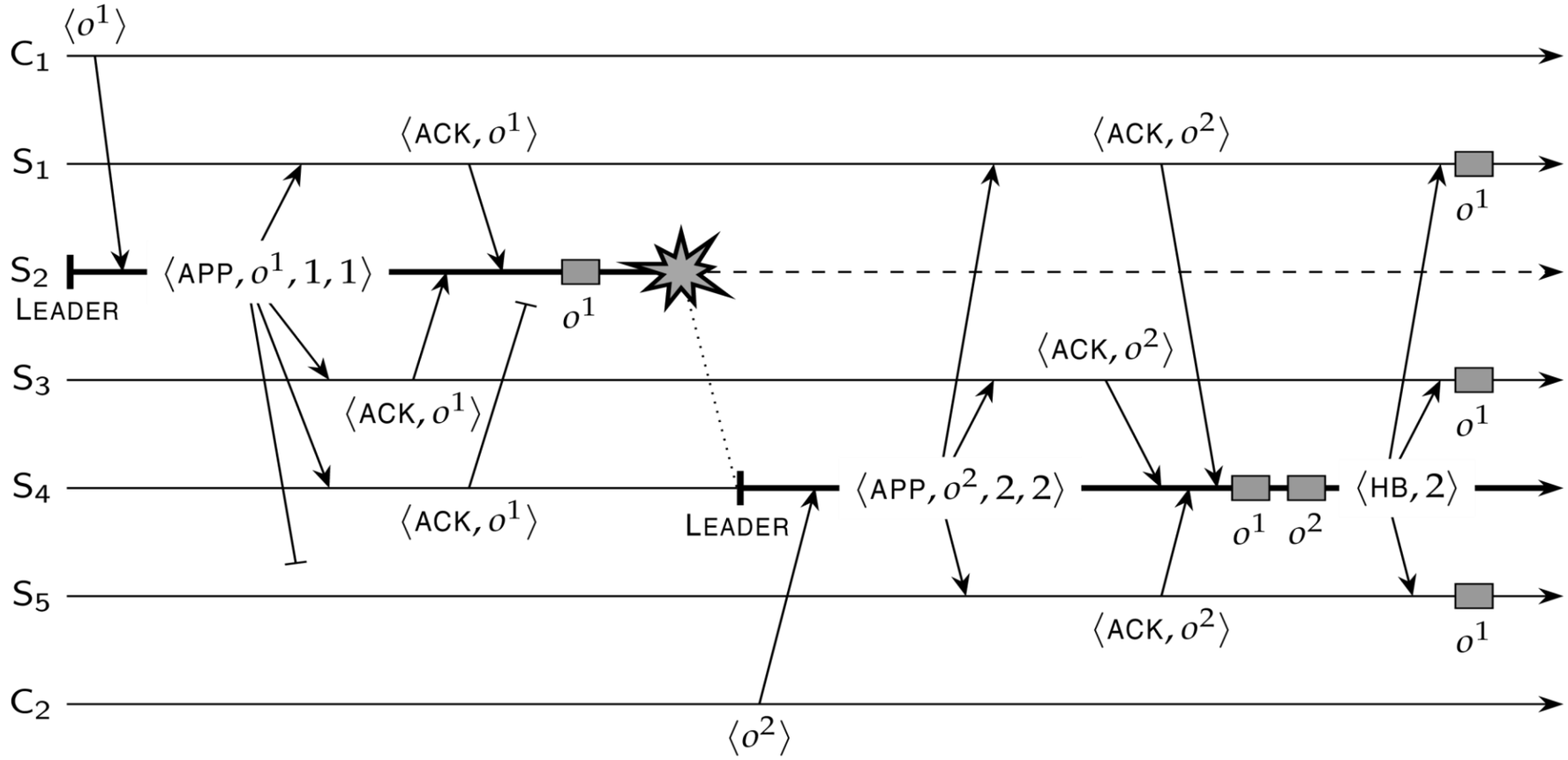


Crucial observations

- We need to understand that the new leader will have a log that includes at least the committed operations from a majority of the remaining servers. If it didn't, it wouldn't have been elected as the new leader. Any missing commits will eventually be sent to the other backups.

Process Resilience

- Consensus in Faulty Systems with Crash Failures: Raft

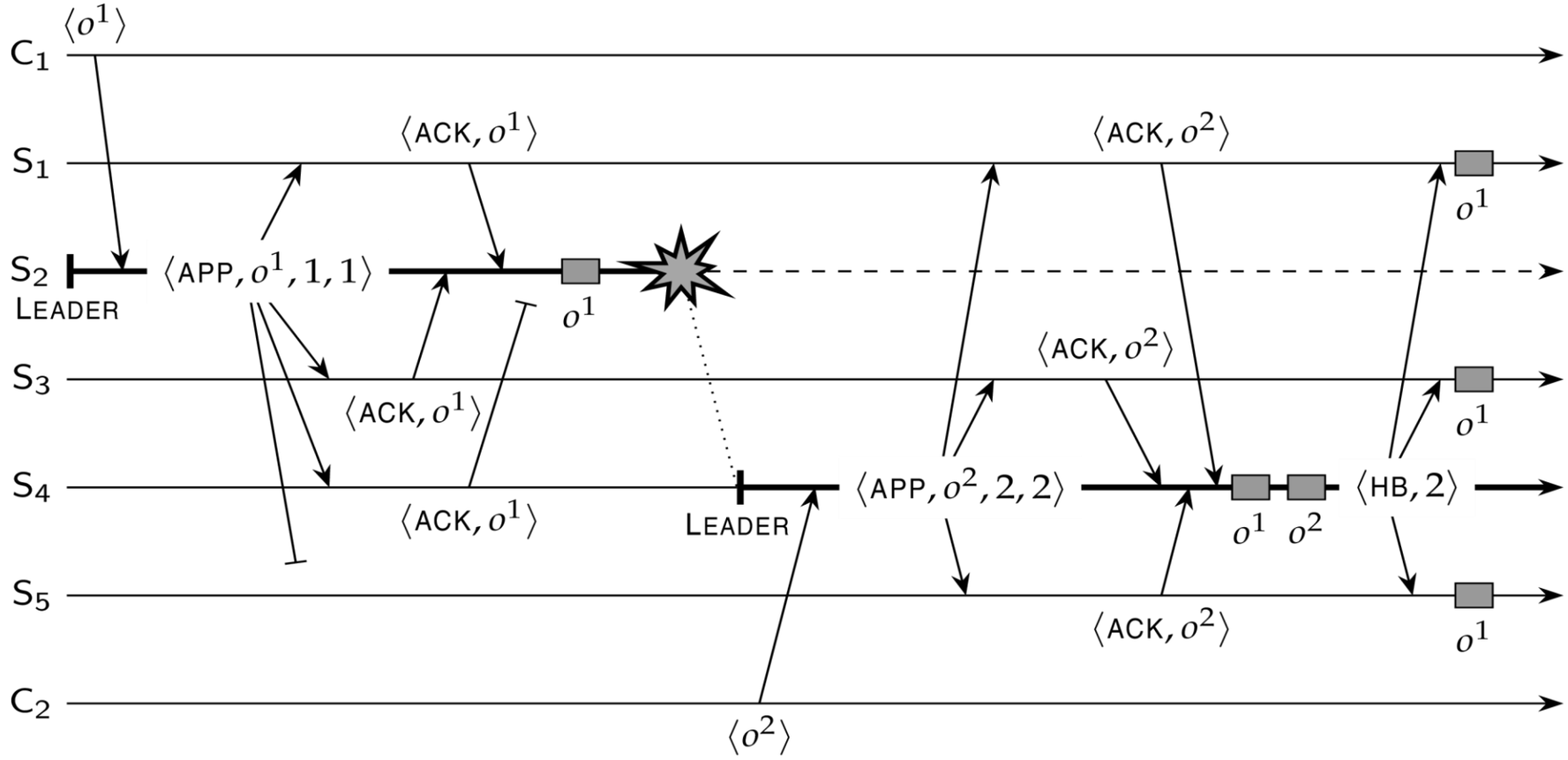


Crucial observations

- When the new leader handles a new operation o2, it broadcasts its log to the other servers. Once it receives acknowledgments from the majority, it can commit o2 and also o1. For example, server S5 will be updated about operation o1 as soon as the new leader tells it about o2. After getting enough acknowledgments, the leader can commit both o1 and o2.

Process Resilience

- Consensus in Faulty Systems with Crash Failures: Raft



Crucial observations

- The other servers will then commit o1 as soon as they see that the leader has done so. We've skipped some details here, especially the fact that the leader doesn't actually send entire logs, but rather just the operation requests. This means that after an election, some servers might be missing operations or have extra ones. Synchronizing the followers' logs with the new leader's log requires some extra steps, but for simplicity, we can assume entire logs are sent.

Process Resilience

- Example: Paxos

  Operate under some 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.

Process Resilience

- Example: Paxos

Types of Processes

- Clients: Request specific operations.
- Proposers: Represent clients on the server side and attempt to get client requests accepted.
- Acceptors: Decide whether to accept proposed operations.
- Learners: Execute the chosen operations once a majority of acceptors agree.
- Typically, there's one leading proposer (the leader) driving the protocol towards consensus. If a majority of acceptors agree on a proposal, it's considered chosen. The learners then execute the chosen proposal once they've been informed by a majority of acceptors.

Process Resilience

- Example: Paxos

Simplified view of how it works:

- The leader receives requests from clients and sends proposals to all acceptors.
- Each acceptor that agrees broadcasts a learn message.
- When a learner receives the same learn message from a majority, it executes the operation.

Key issues:

- Reaching and Ensuring Execution: It's not enough to just reach consensus; we must ensure that a majority of non-faulty servers actually execute the operation. This is handled by retransmitting learn messages and having acceptors log their decisions. This helps detect missing messages and ensures operations are executed in the same order.
- Handling a Failing Leader: If the leader fails, ideally, a new leader is elected, and the failed leader recognizes the change upon recovery. However, Paxos tolerates multiple proposers thinking they're the leader simultaneously. This requires distinguishing between proposals to ensure only the current leader's proposals are accepted. Hence, a leader-election algorithm is necessary, often integrated into Paxos.

Process resilience

- Example: Paxos

Observation:

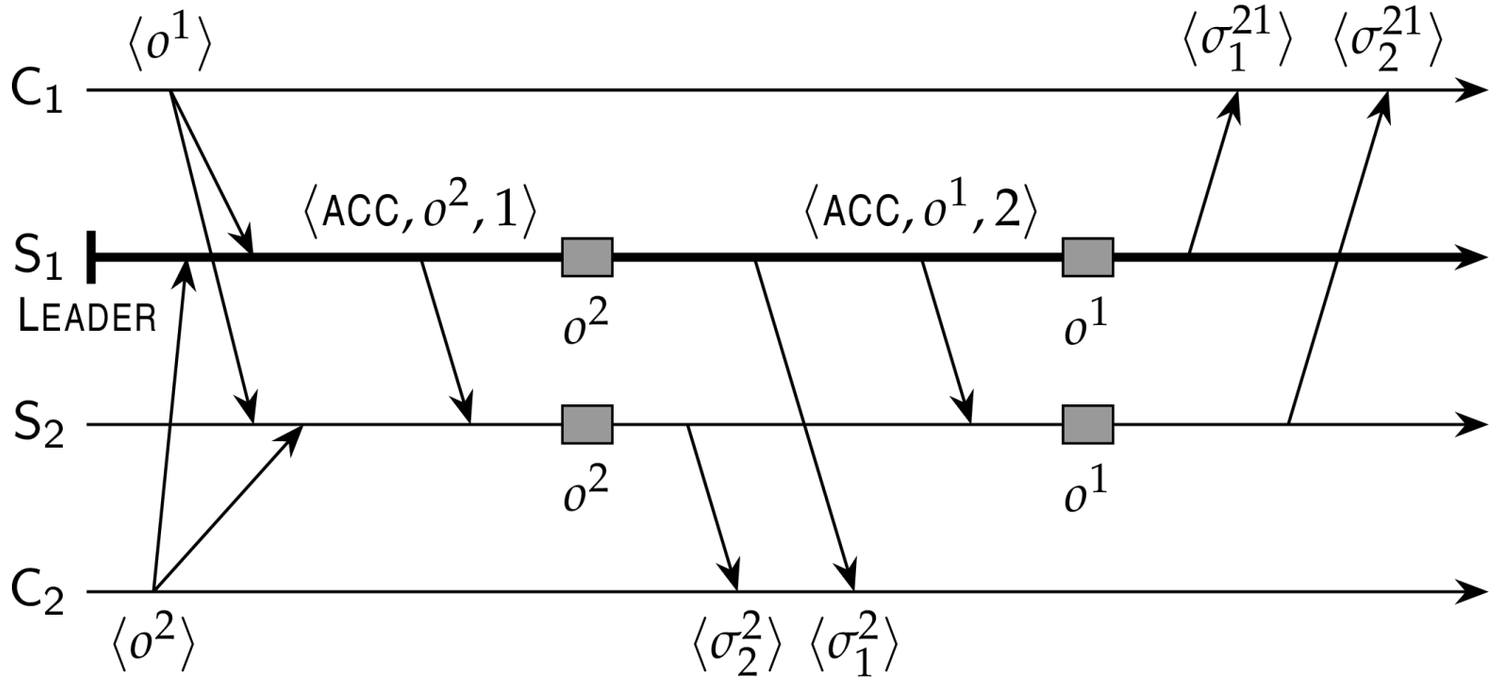
- Imagine we start with a server that we want to make more reliable. We know we can do this by replicating the server and ensuring all commands from clients are executed by all servers in the same order. The simplest way to do this is to add another server, so now we have two servers, S1 and S2. To keep the commands in order, we make one server the sequencer. This sequencer assigns a unique timestamp to every command submitted, and the servers execute commands based on these timestamps. In Paxos, this sequencer is called the leader. You can think of it as the primary server, with the other one acting as a backup.

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.

Process Resilience

- Example: Paxos

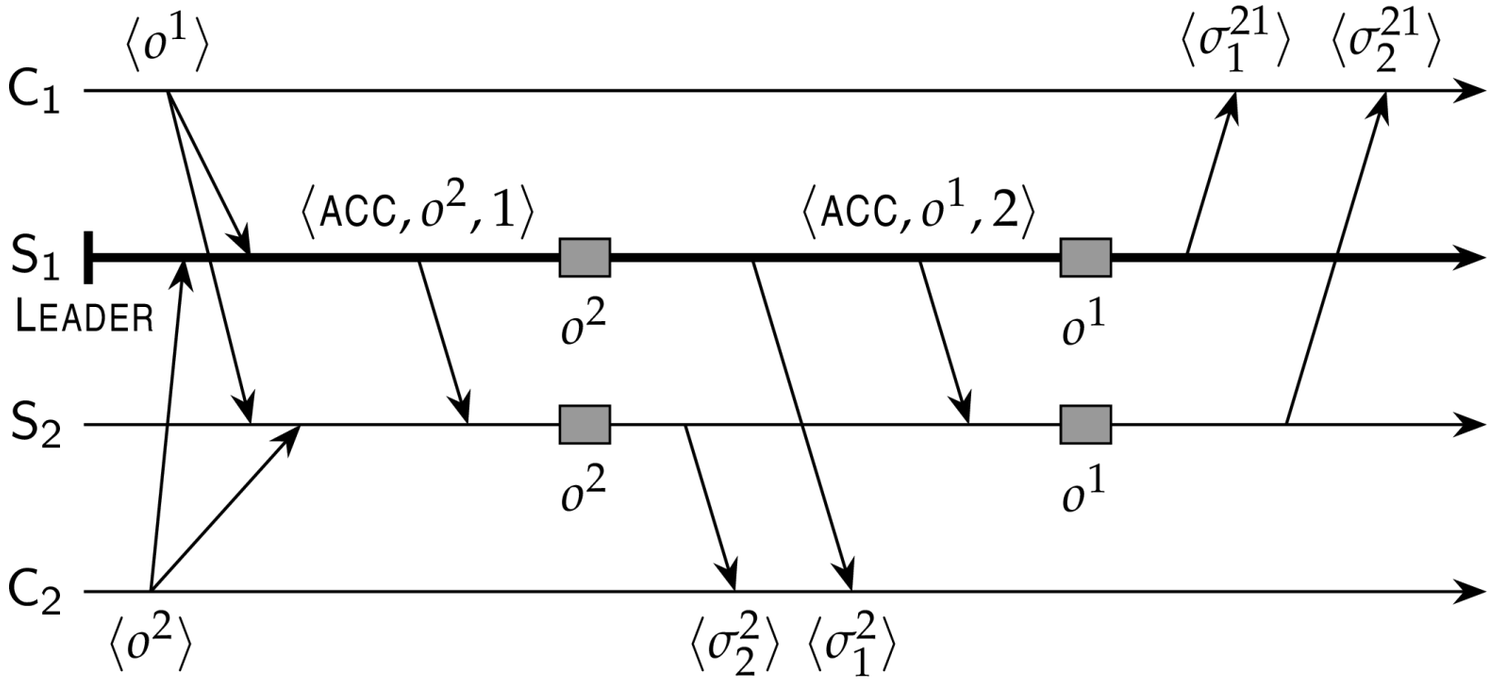


Observation:

- Imagine a client sends a command to all servers. If a server misses a command, it can get it from another server.
- Assume all commands are stored at the servers. The main thing we need to focus on is making sure the servers agree on which command to execute next. This means the servers only need to communicate control messages with each other.

Process Resilience

- Example: Paxos



Observation:

- Server S1 is the leader. As the leader, S1 assigns timestamps to the commands. Client C1 sends command o1, and client C2 sends command o2. S1 tells S2 to execute command o2 first with timestamp 1, and then execute command o1 with timestamp 2. After a server processes a command, it sends the result back to the client. We use the notation ⟨σj i⟩ to show this, where i is the server's index and j is the state it was in, represented by the sequence of operations it has performed. So, in our example, client C1 will get the result ⟨σ21⟩, indicating that the second server has executed command o1 after executing command o2.

Process Resilience

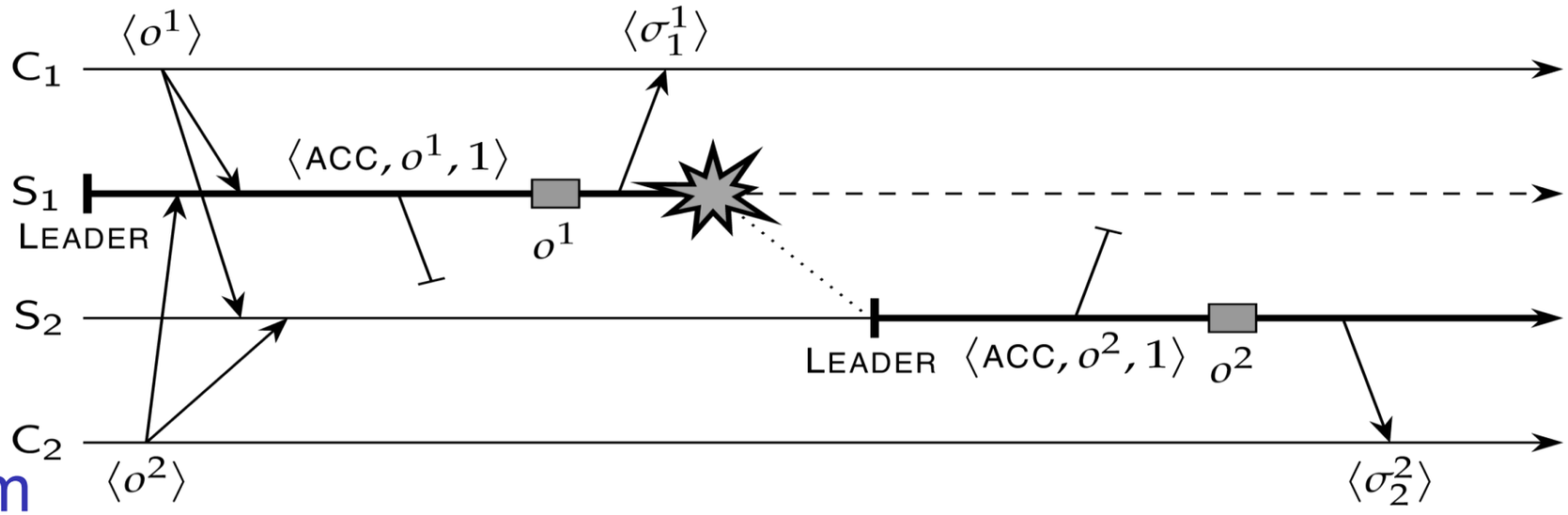
- Example: Paxos

Some Paxos terminology

- The leader sends an accept message ACCEPT(o, t) to backups when assigning a timestamp t to command o.
- A backup responds by sending a learn message: LEARN(o, t)
- When the leader notices that operation o has not yet been learned, it retransmits ACCEPT(o, t) with the original timestamp.

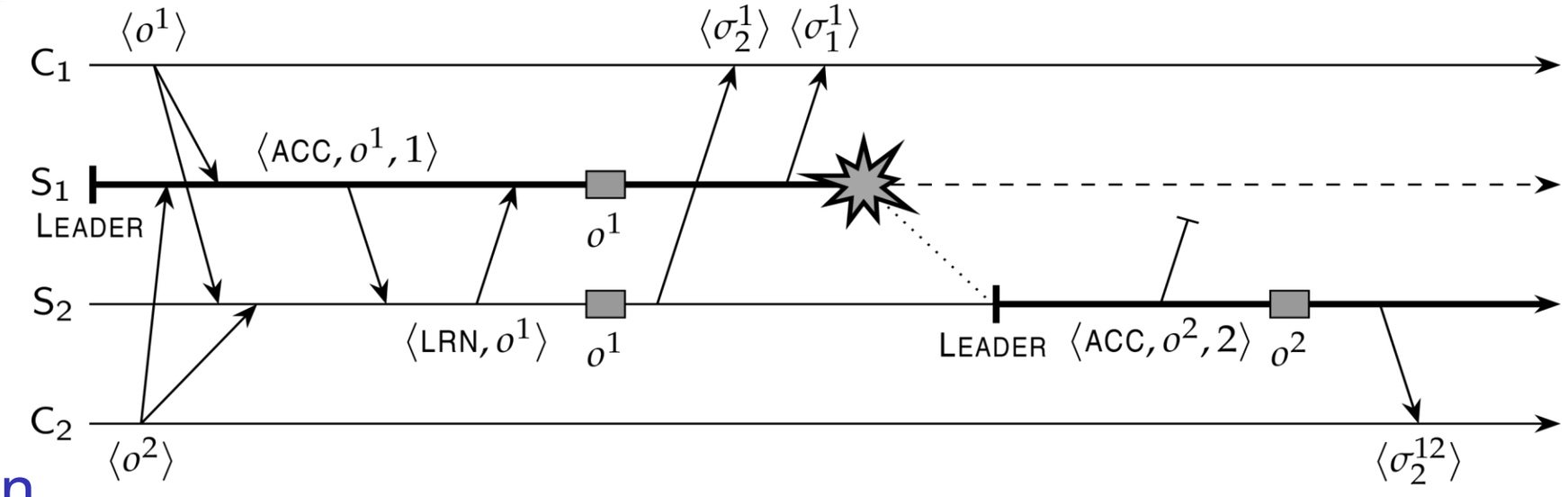
Process Resilience

- Example: Paxos



Problem

Primary crashes after executing an operation, but the backup never received the accept message. The problem here is that server S2 will never learn about operation o1.

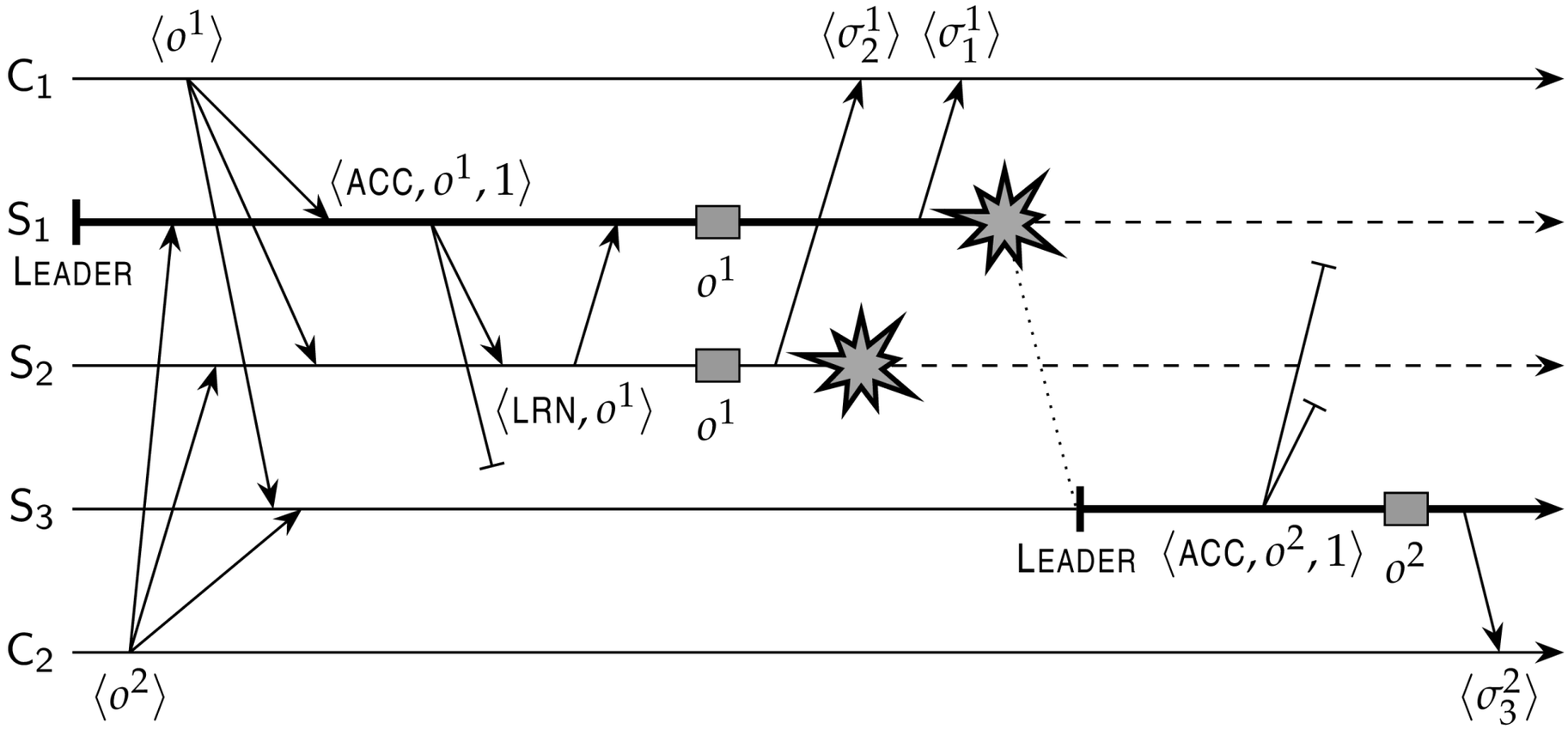


Solution

A server only executes an operation if it knows the other server has also learned about the operation

Process Resilience

- Example: Paxos



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. This means S2 also needs to wait to execute o1 until it knows that S3 has learned about the operation.

Process Resilience

- Example: Paxos

General rule

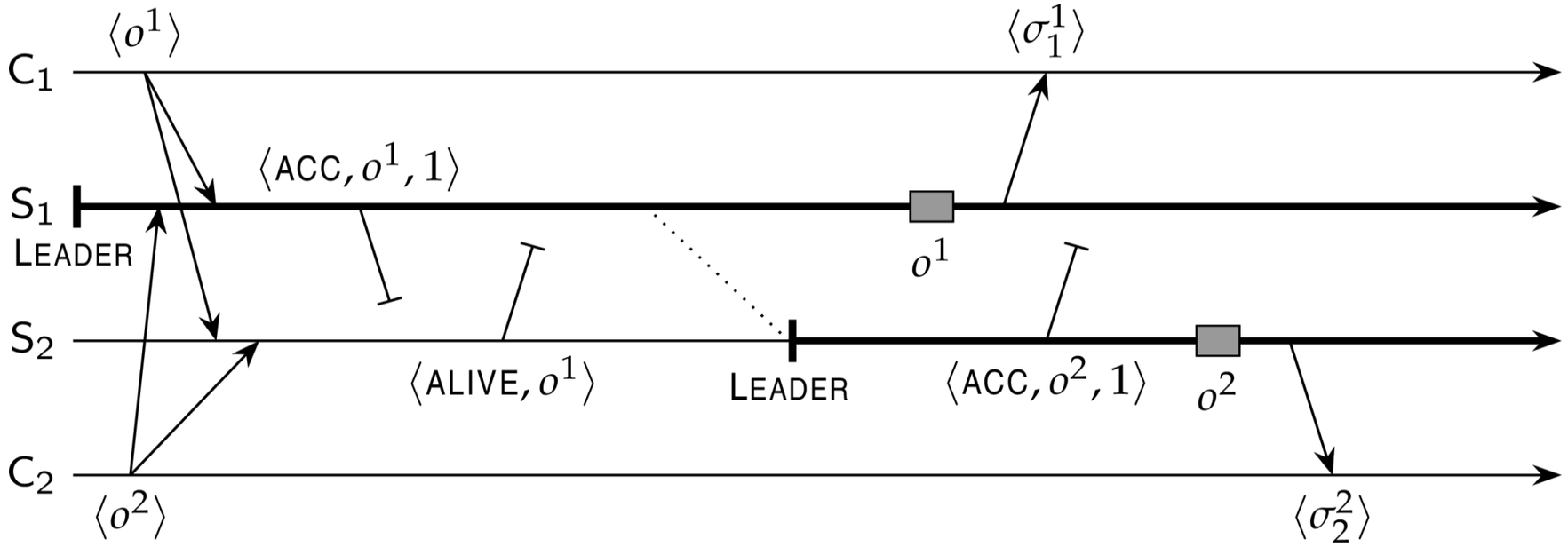
In Paxos, a server (let's call it S) cannot execute an operation (o) until it has received a learn(o) message from all other non-faulty servers. So far, we've assumed that a server can reliably detect when another server has crashed. Actually, this isn't always the case. Typically, to detect failures, we use timeouts on expected messages. For example, each server must regularly send a message indicating it's still alive. Other servers set timeouts for receiving these messages. If a timeout expires, they suspect the sender has failed. In partially synchronous or fully asynchronous systems, this method is the only option. However, this can lead to false detections because the "I'm alive" messages might just be delayed or lost.

Process Resilience

- Example: Paxos

Practice

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



Problem and Solution

Each server might independently decide to execute its chosen operation, causing inconsistent behavior. To fix this, we introduce an extra server and require that a server can only execute an operation if it knows a majority will execute it too. In a three-server setup, a server (let's call it S) can execute an operation (o) as soon as it receives at least one other learn(o) message. With the sender of that message, S will form a majority and ensure consistency.

Process Resilience

- Example: Paxos

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) LEARN(*o*) message, so that it knows that a majority of servers will execute *o*.

Process Resilience

- Example: Paxos

If one of three servers crashes, assumptions before taking the next steps

- Initially, S_1 is the leader.
- A server can reliably detect it has missed a message, possibly using timestamps or strictly increasing sequence numbers. If a server notices a missed message, it can request a retransmission to catch up before continuing.
- When a new leader needs to be elected, the remaining servers follow a strictly deterministic algorithm, if S1 crashes, S2 will become the leader. If S2 crashes, S3 takes over, and so on.
- Clients might receive duplicate responses, but they only need to recognize these duplicates and don't play a role in resolving issues within the Paxos protocol. In other words, clients don't help servers resolve conflicts.

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.

Process Resilience

- Example: Paxos

Observation

S1, the leader, crashes after executing operation o1. The worst-case scenario here is that S3 doesn't know about the crash until the new leader, S2, tells it to accept operation o2. When S2 sends the accept(o2,2) message with timestamp t = 2, S3 will realize it missed an earlier message. S3 will notify S2, which will then resend accept(o1,1) so S3 can catch up.

S_3 is completely ignorant of any activity by S_1

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

Process Resilience

- Example: Paxos

Observation

Similarly, if S2 missed accept(o1,1) but detected that S1 crashed, it will eventually send either accept(o1,1) or accept(o2,1) to S3, both with the same timestamp t = 1. This gives S3 enough information to get S2 back on track. If S2 sent accept(o1,1), S3 can tell S2 it already learned o1. If S2 sent accept(o2,1), S3 will inform S2 that it missed operation o1. So, even if S1 crashes after executing an operation, Paxos still works correctly.

S_2 missed ACCEPT(o^1, 1)

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

Process Resilience

- Example: Paxos

S_3 is completely ignorant of any activity by S_1

What if S1 crashes right after sending accept(o1,1) to the other two servers? If S3 is unaware because messages are lost, S2 will take over and announce that o2 should be accepted. Just like before, S3 can tell S2 it missed operation o1, and S2 will help S3 catch up.

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 ACCEPT(o^1, 1)

If S2 misses messages but detects S1's crash, it will use an outdated timestamp when it proposes a new operation. This will prompt S3 to inform S2 that it missed operation o1, resolving the issue.

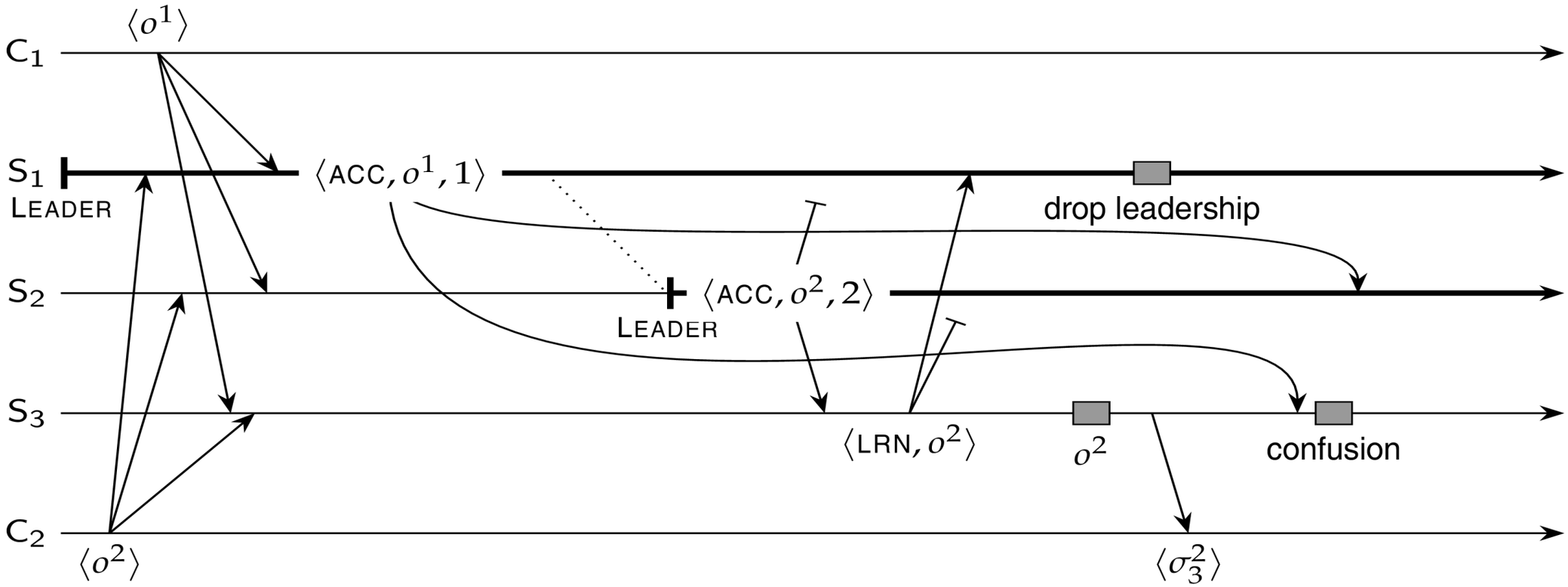
As soon as S_2 proposes an operation, it will be using a stale timestamp, allowing S_3 to tell S_2 that it missed operation o^1.

Observation

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

Process Resilience

- Example: Paxos

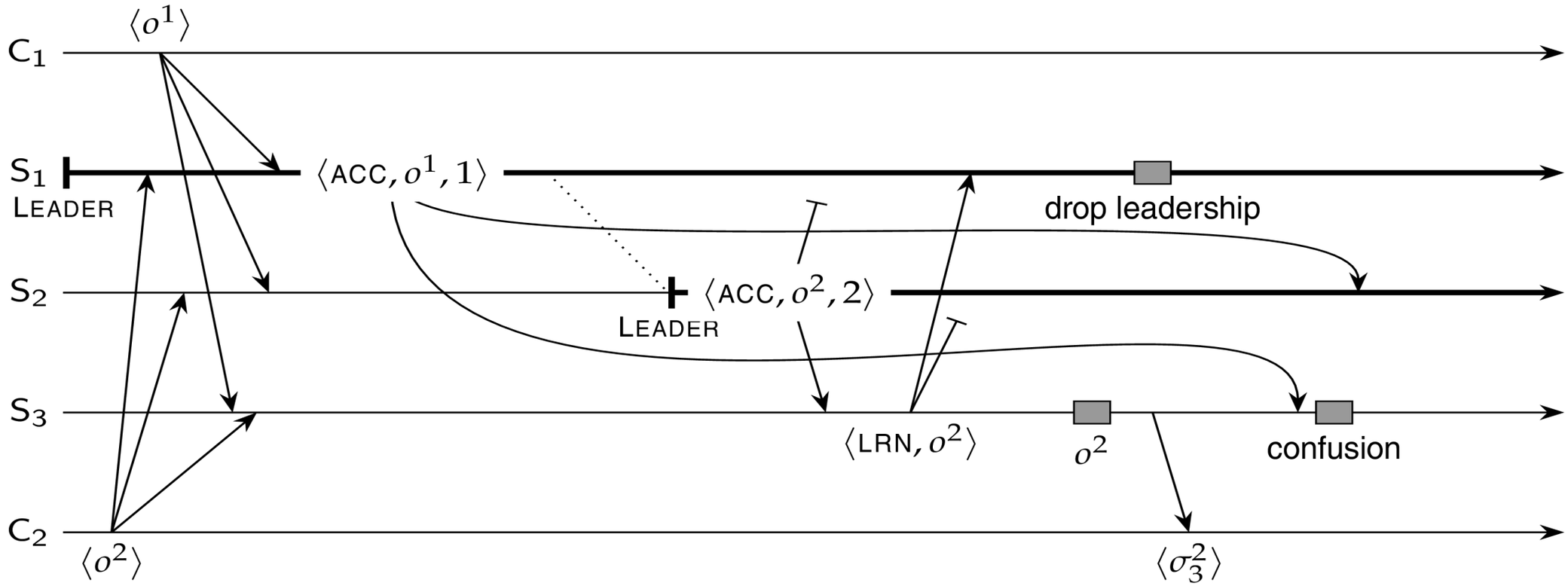


Problem and solution

The accept messages from S1 are delayed, and S2 wrongly thinks S1 has crashed. S2 takes over as the leader and sends accept(o2,1) with a timestamp of t = 1. However, when the delayed accept(o1,1) finally arrives, S3 is confused because it wasn't expecting that message. S3 doesn't check who the current leader is; it only knows that it's not the leader, so it doesn't expect an accept message with the same timestamp again.

Process Resilience

- Example: Paxos

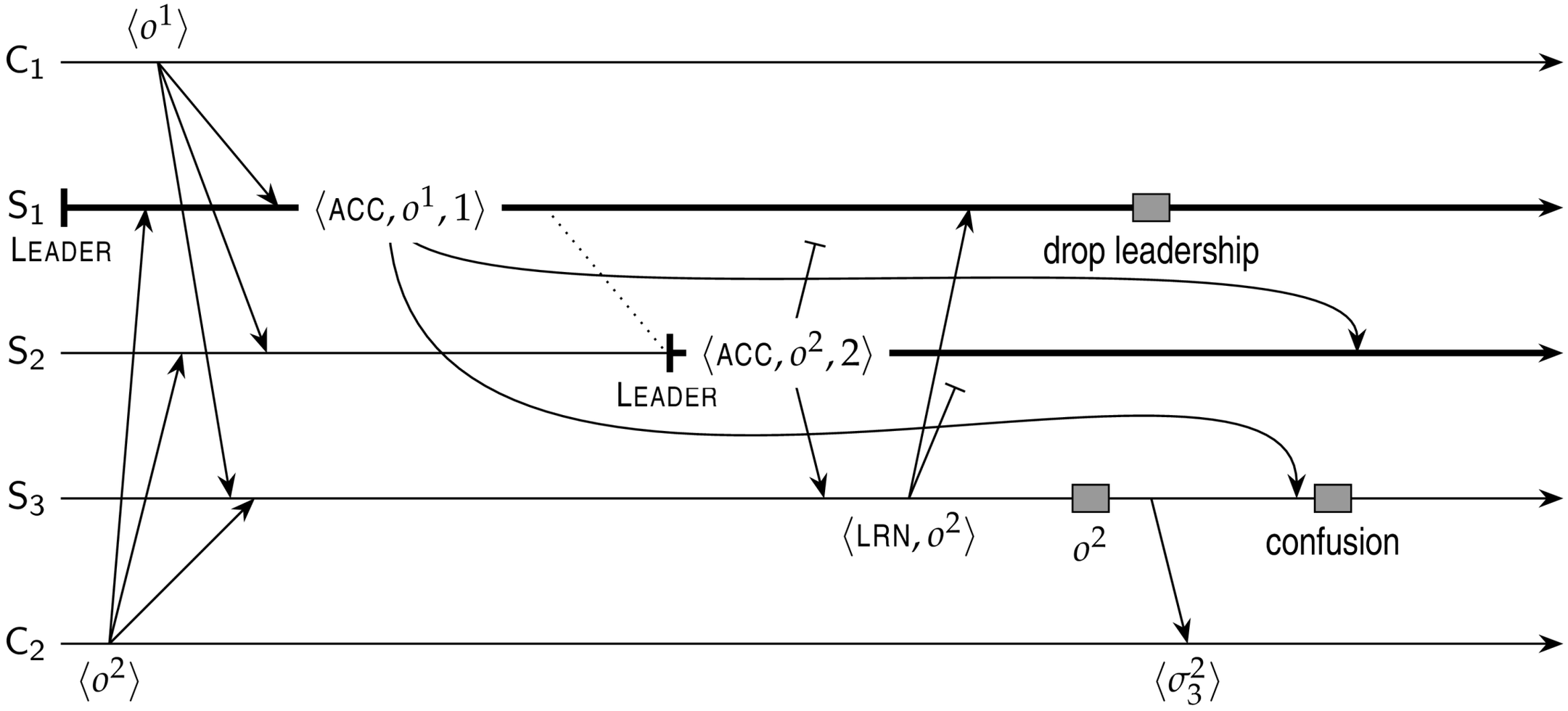


Problem and solution

Things would be different if S3 knew who the current leader was and there was a clear way to elect a new leader. In that case, S3 could safely reject accept(o1,1) because it would know that S2 is now the leader. S3 could even resend learn(o2) to S1. If S1 gets a learn(o2) message, it would realize that leadership has been taken over. The key point here is that the leader should include its ID in the accept message to avoid this confusion.

Process Resilience

- Example: Paxos

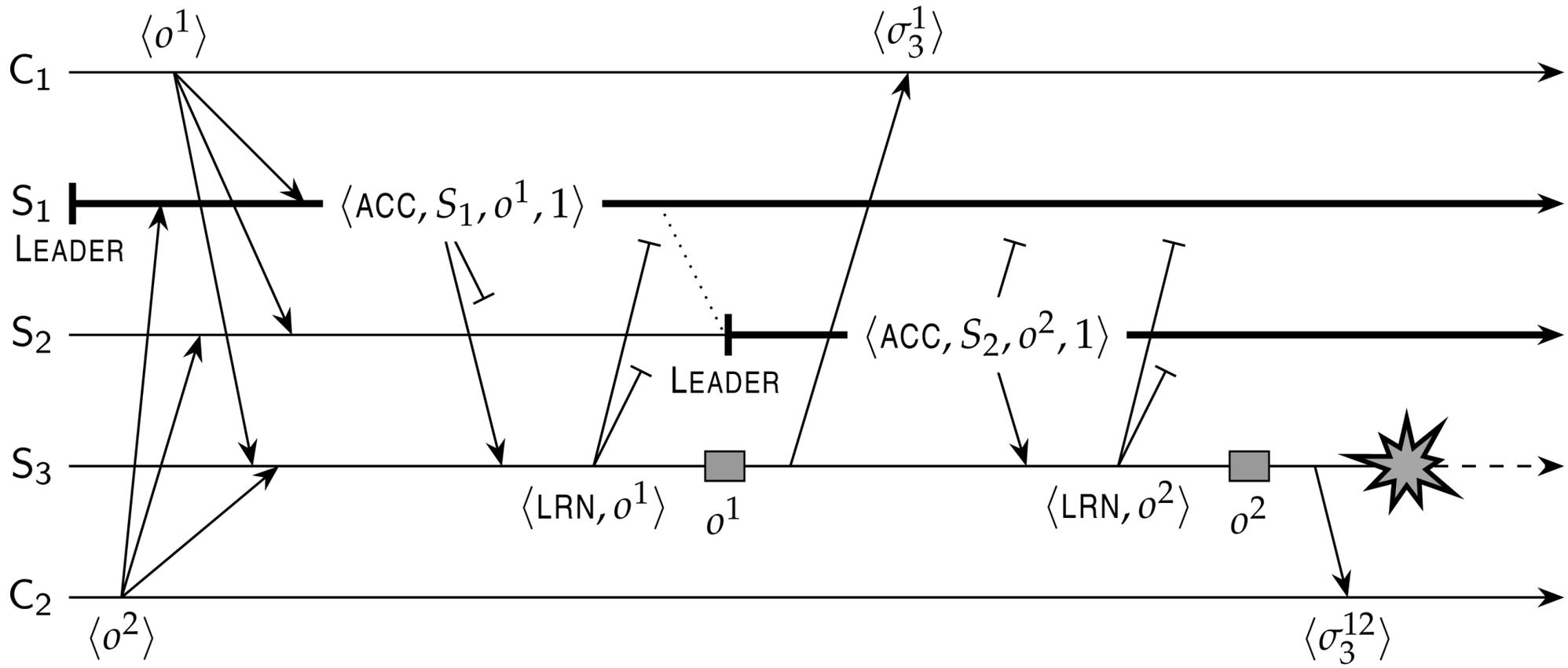


Problem and solution

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

Process Resilience

- Example: Paxos



Essence of solution

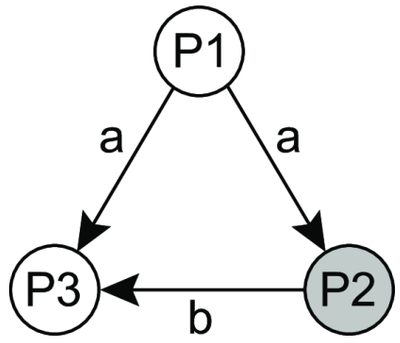
When S_2 takes over, it needs to make sure that any outstanding operations initiated by S_1 have been properly flushed, i.e., executed by enough servers. This requires an explicit leadership takeover by which other servers are informed before sending out new accept messages.

Process Resilience

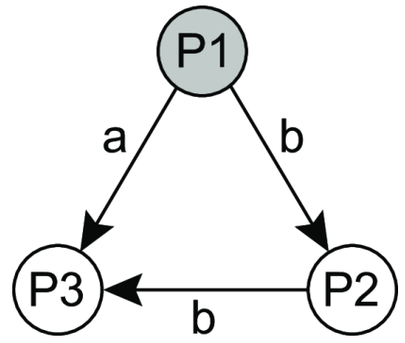
- Consensus in Faulty Systems with Arbitrary Failures

Essence

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



Improper forwarding



Different messages

Improper forwarding: Process P2 forwarding a different value or operation than it's supposed to. In Paxos terms, this could mean a primary server telling the backups that a different operation (o′) was accepted instead of the correct one (o).
Different messages: P1 is telling different things to different processes, like sending operation o to some backups and operation o′ to others. These aren't necessarily malicious actions; they could just be omission or commission failures.

Observations

Reaching consensus in a fault-tolerant process group where k members can fail, assuming arbitrary failures. We will show that to reach consensus under these failure conditions, we need at least 3k + 1 members.

Process Resilience

- Consensus in Faulty Systems with Arbitrary Failures
  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 the primary 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

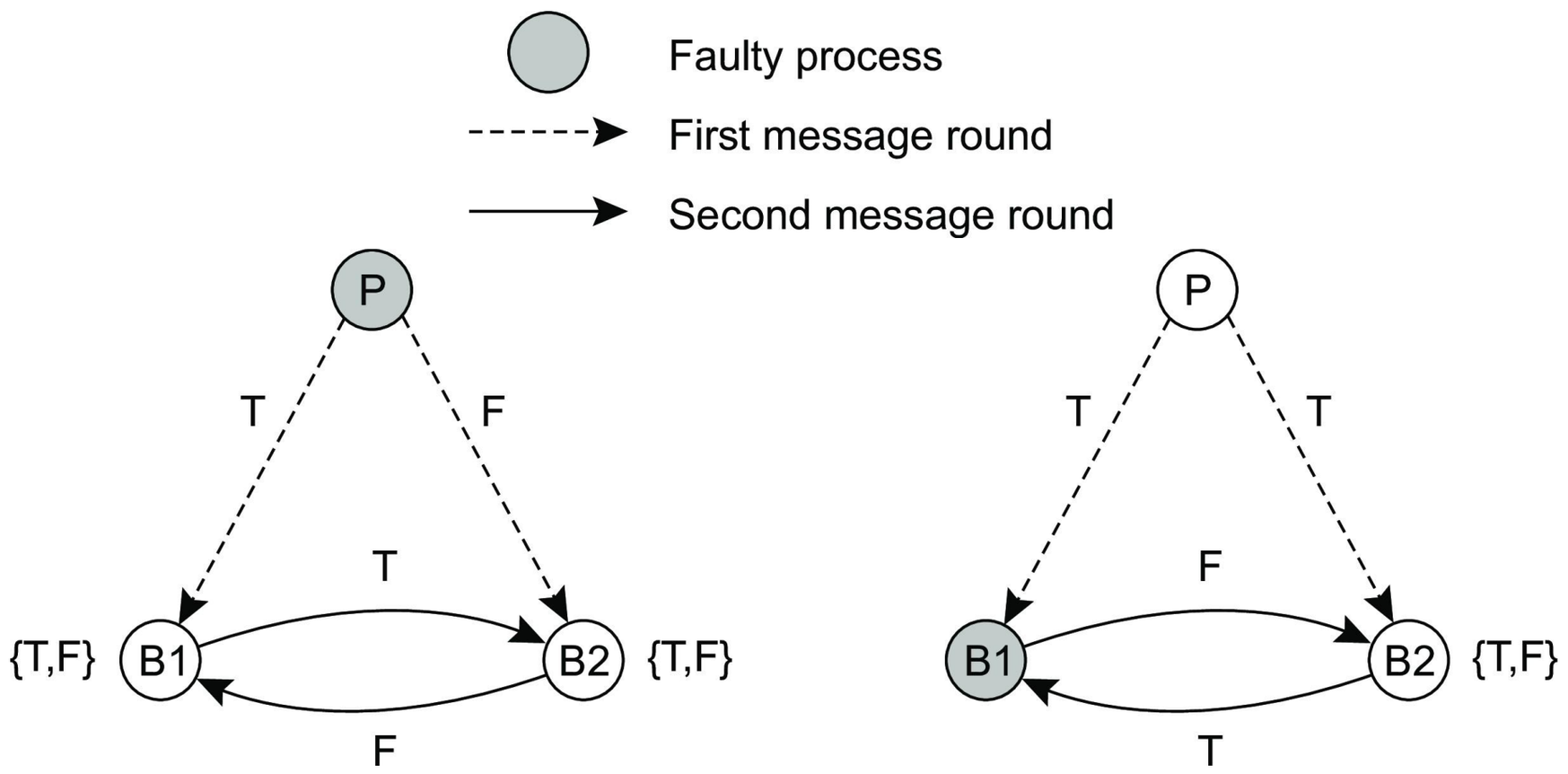
BA1: Every nonfaulty backup process stores the same value.
BA2: If the primary is nonfaulty then every nonfaulty backup process stores exactly what the primary had sent.

Observation

- Primary faulty ⇒ BA1 says that backups may store the same, but different (and thus wrong) value than originally sent by the client.
- Primary not faulty ⇒ satisfying BA2 implies that BA1 is satisfied.

Process Resilience

- Consensus in Faulty Systems with Arbitrary Failures

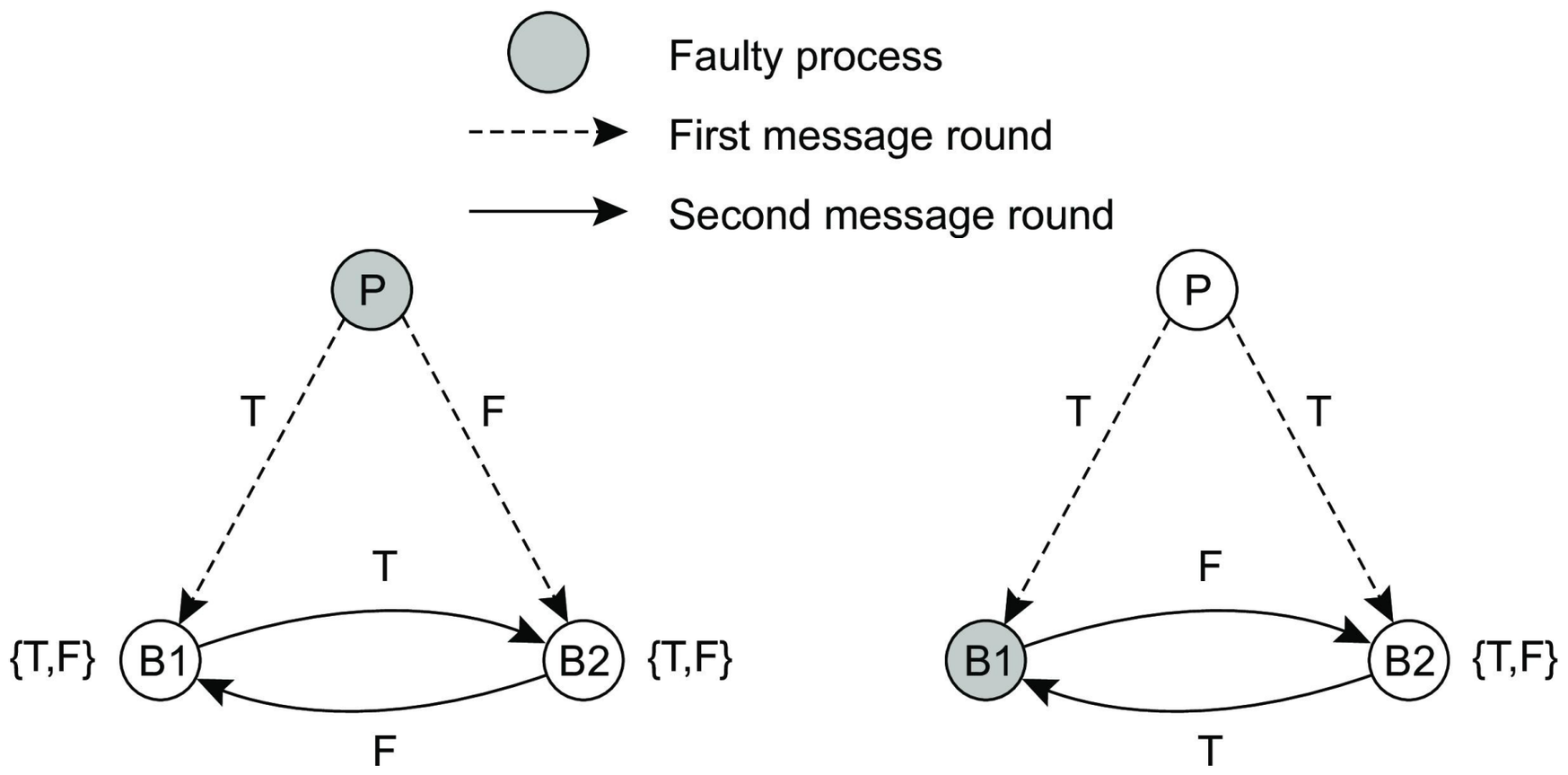


To tolerate the failure of a single process

The faulty primary (P) sends two different values to the backups B1 and B2. To reach consensus, both backups forward the received value to each other, leading to a second round of message exchanges. At this point, B1 and B2 each have received the set of values {T, F}, making it impossible to come to a conclusion. Similarly, consensus can't be reached when incorrect values are forwarded.

Process Resilience

- Consensus in Faulty Systems with Arbitrary Failures

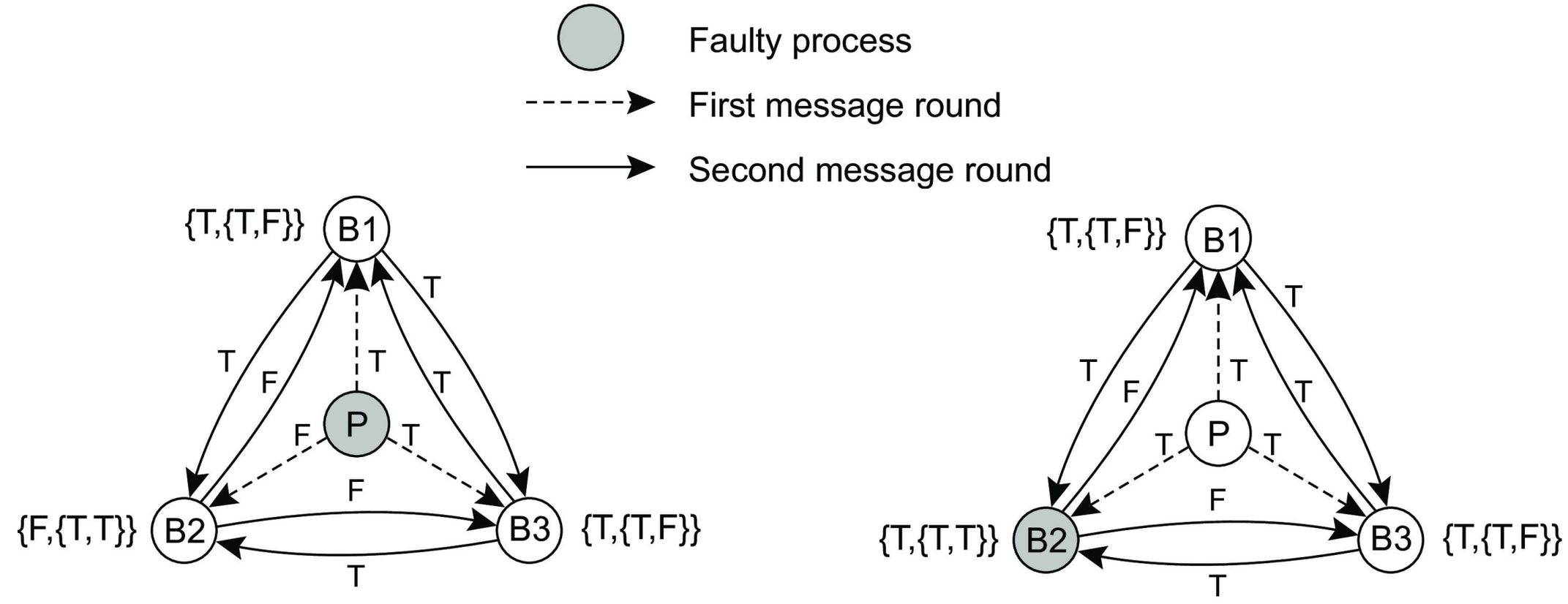


To tolerate the failure of a single process

The primary (P) and backup B2 operate correctly, but B1 does not. Instead of forwarding the value T to B2, it sends the incorrect value F. As a result, B2 now has the set of values {T, F} and can't draw any conclusions. This means P and B2 can't reach consensus, and B2 can't decide what to store, failing to meet requirement BA2.

Process Resilience

- Consensus in Faulty Systems with Arbitrary Failures

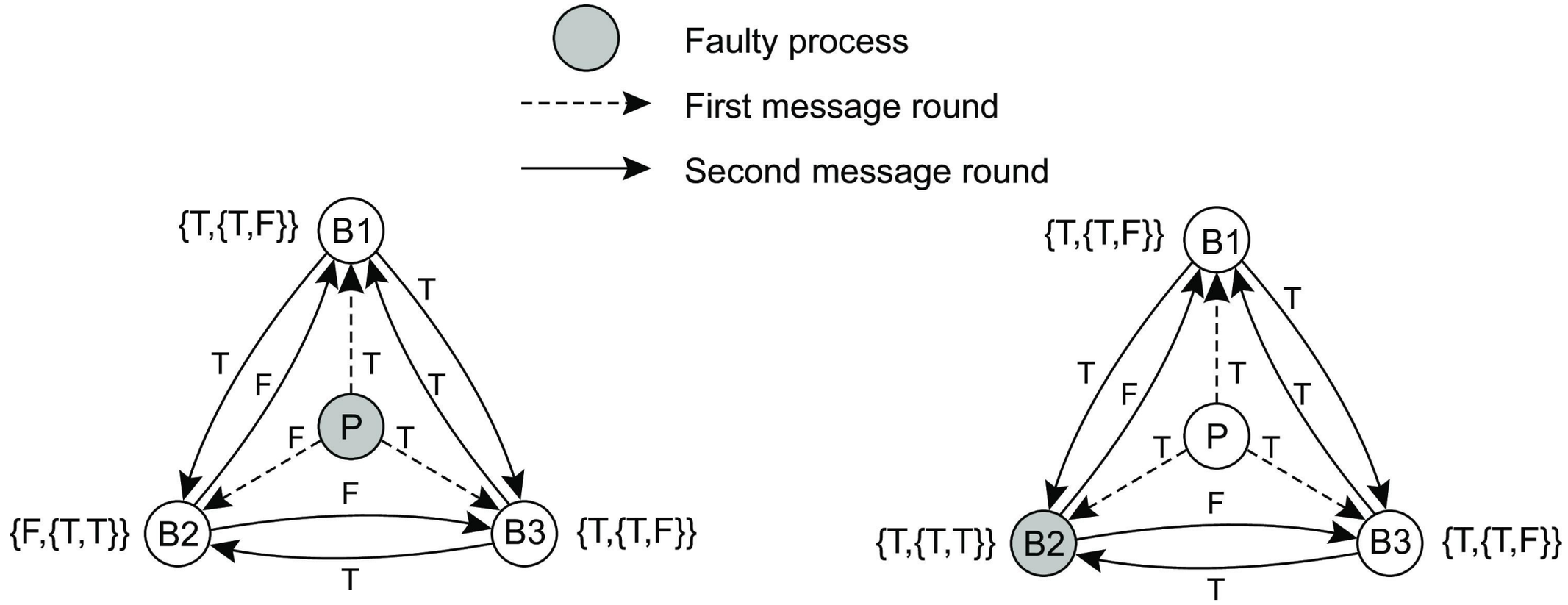


A group of 3k+1 processes

Even if k members fail arbitrarily, the remaining non-faulty processes can still reach consensus. We'll start with the case where n = 4 (k = 1). There are one primary (P) and three backups (B1, B2, and B3). The primary (P) is faulty and provides inconsistent information to its backups. In our solution, processes will forward what they receive to the others. In the first round, P sends T to B1, F to B2, and T to B3. Each backup then forwards what they received to the others. After two rounds, each backup will have received the set {T, T, F}, and they can reach consensus on the value T.

Process Resilience

- Consensus in Faulty Systems with Arbitrary Failures



The case where one backup fails

Let's assume the non-faulty primary sends T to all backups, but B2 is faulty. B1 and B3 will send T to the others in the second round, while B2 might send F. Despite this failure, B1 and B3 will conclude that P sent T, meeting our requirement BA2. This demonstrates that with 3k+1 processes, we can still reach consensus even if up to k members fail arbitrarily.

Process Resilience

- Consensus in Faulty Systems with Arbitrary Failures: Practical Byzantine Fault Tolerance

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

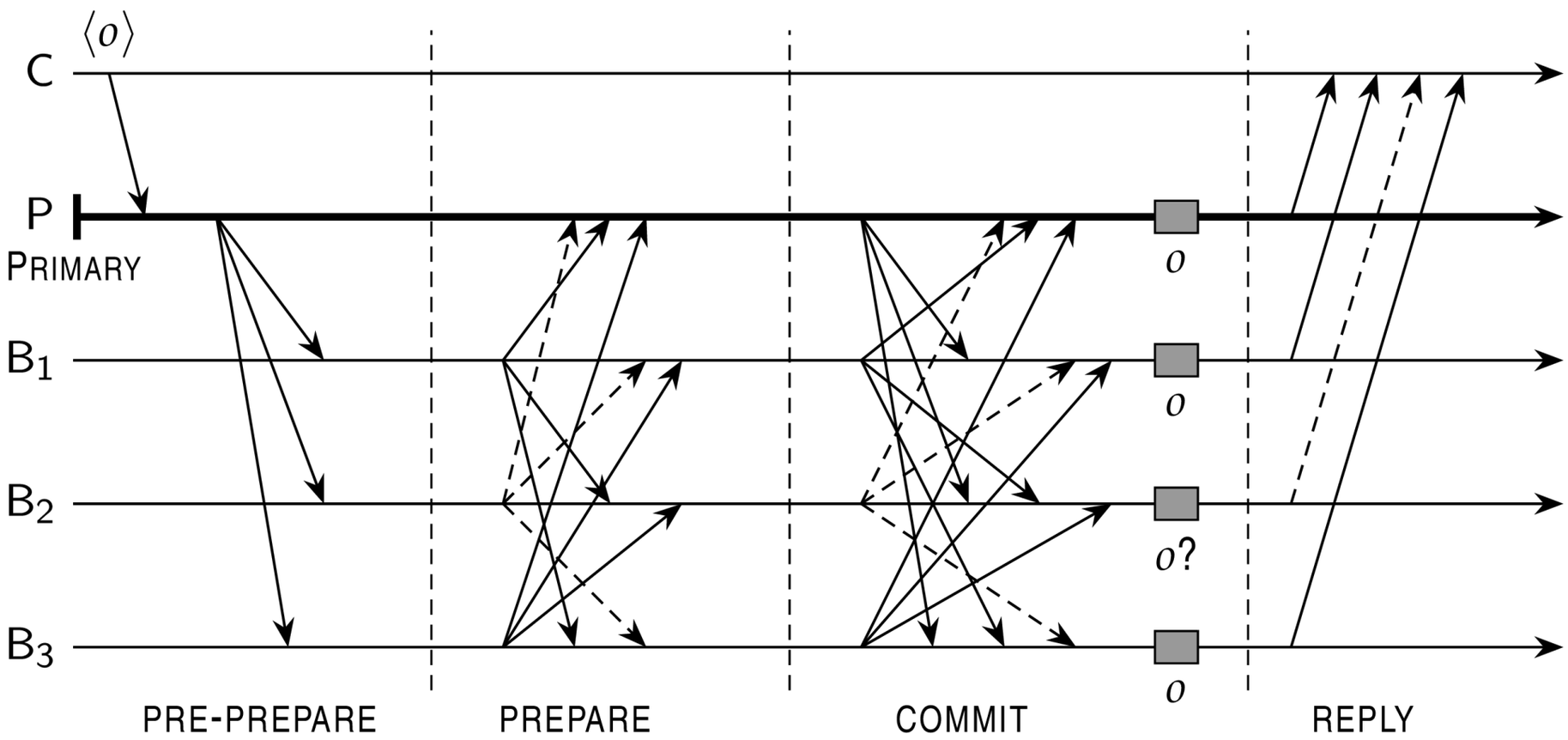
- A server may exhibit arbitrary failures
- Messages may be lost, delayed, and received out of order
- Messages have an identifiable sender (i.e., they are signed)
- Partially synchronous execution model

Essence

A primary-backup approach with $3k + 1$ replica servers.

Process Resilience

- Consensus in Faulty Systems with Arbitrary Failures: Practical Byzantine Fault Tolerance

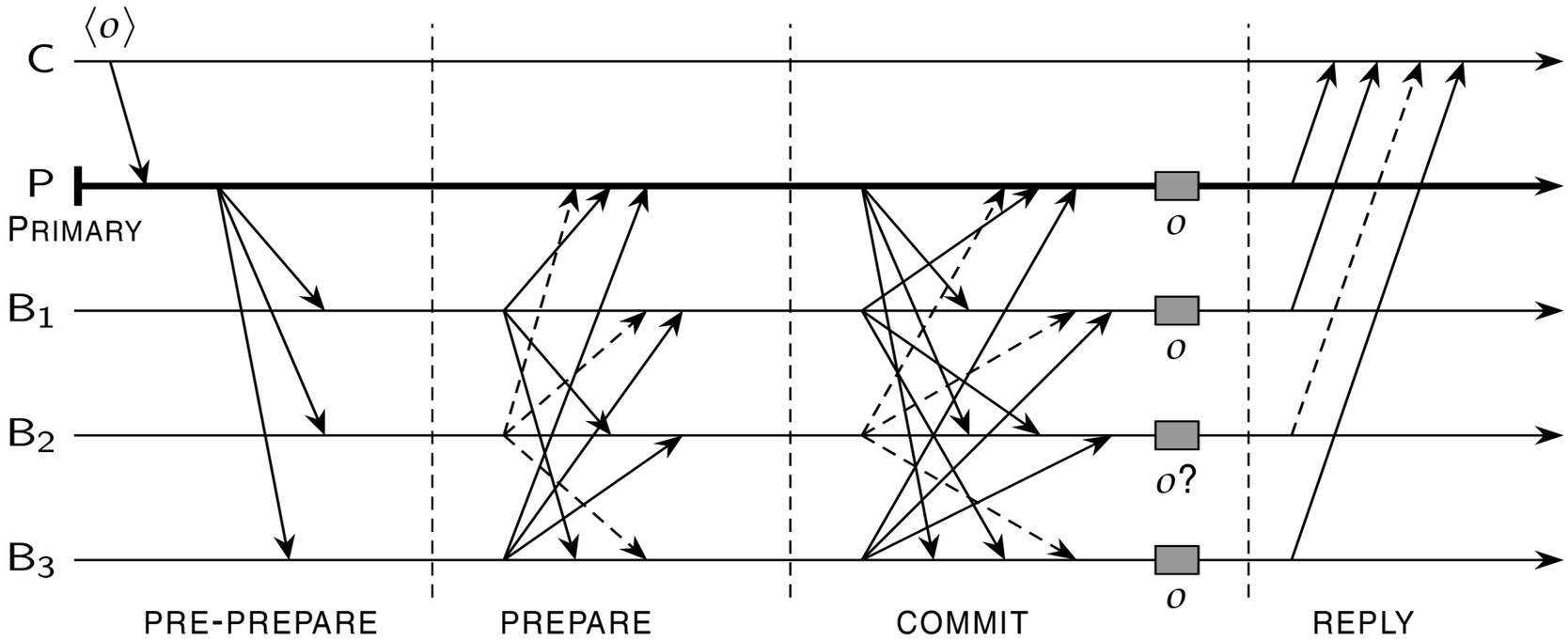


- *C* is the client
- *P* is the primary
- B_1, B_2, B_3 are backups
- Assume B_2 is faulty

- 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 is in *v* and has not accepted a an operation with timestamp *t* before.

Process Resilience

- Consensus in Faulty Systems with Arbitrary Failures: Practical Byzantine Fault Tolerance

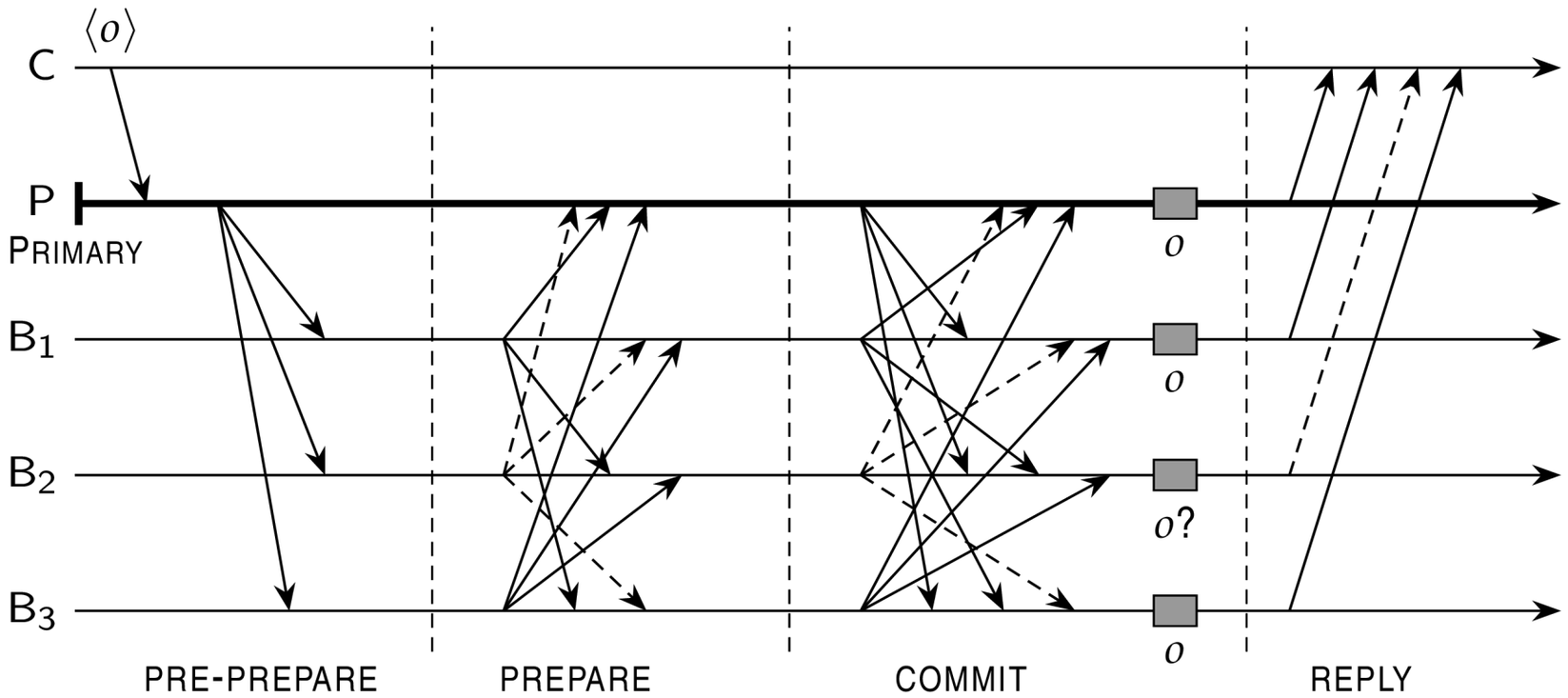


- *C* is the client
- *P* is the primary
- B_1, B_2, B_3 are backups
- Assume B_2 is faulty

- A client (C) sends a request to execute operation (o) to the primary server (P).
- The primary knows which replica servers are currently non-faulty, which it keeps track of with a number called a view (v).
- The primary assigns a timestamp (t) to the operation (o). This timestamp is incremented for each new request.
- The primary then sends a signed pre-prepare message, pre-prepare(t, v, o), to the backup servers (Bi).

Process Resilience

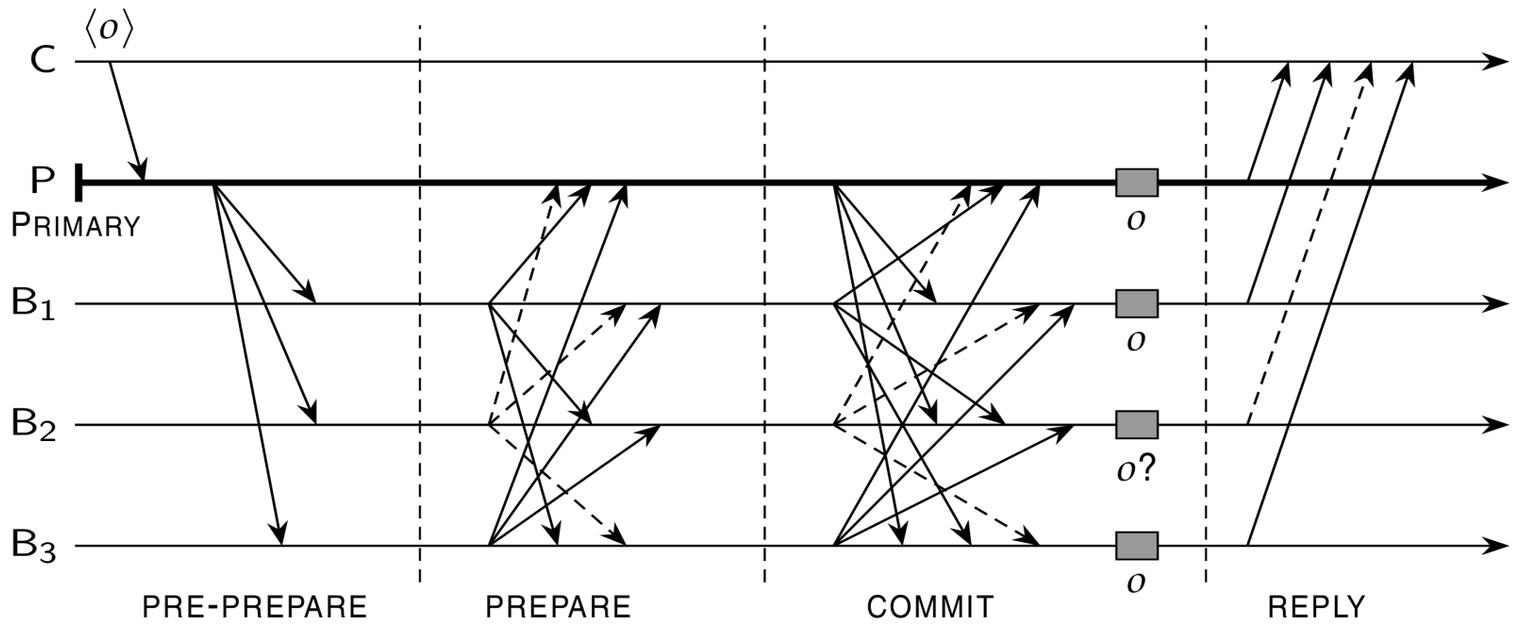
- Consensus in Faulty Systems with Arbitrary Failures: Practical Byzantine Fault Tolerance



- A non-faulty backup will accept a pre-prepare message if it is in the current view (v) and hasn't accepted an operation with the same timestamp (t) in that view before.
- Each backup that accepts the pre-prepare message then sends a signed prepare(t, v, o) message to the others, including the primary.
- Key point: a nonfaulty server will eventually log *2k* messages PREPARE(*t, v, o*) (including its own) that match the pre-prepare message from the primary, there is consensus among the non-faulty servers on the order of the operation..

Process Resilience

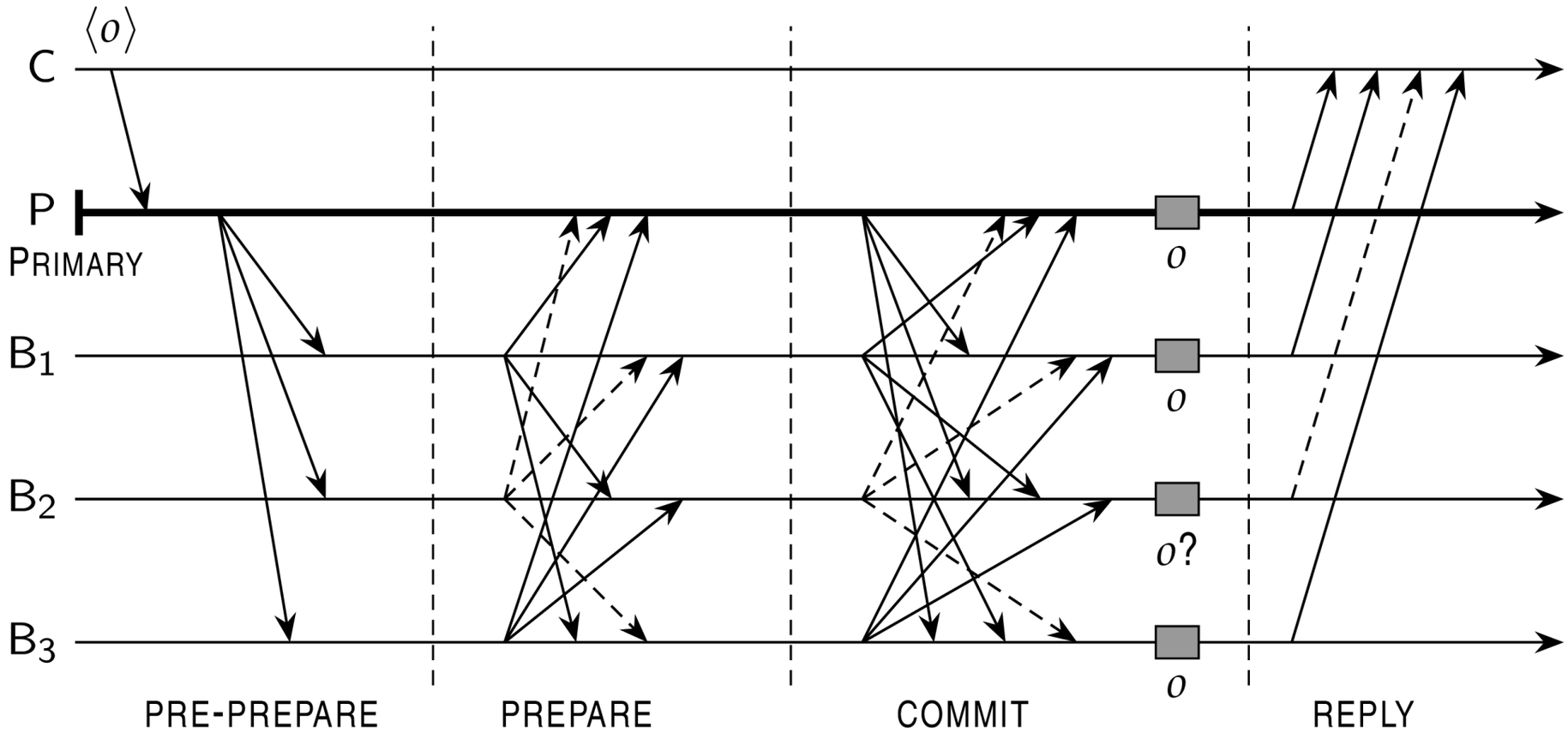
- Consensus in Faulty Systems with Arbitrary Failures: Practical Byzantine Fault Tolerance



- Consider a prepare certificate PC(t, v, o), which is based on a set of 2k+1 messages. If there were another prepare certificate PC(t, v, o') with a different operation o', the intersection of the two certificates would include messages from at least k+1 servers.
- Among these, at least one is non-faulty and will have sent the same prepare message. Therefore, o must equal o'.
- Note: no matter what B2 does, the two non-faulty backups (B1 and B3) will each log three messages for ⟨t, v, o⟩: the original message from P, their own message, and the message from the other backup. At this point, there is consensus to execute operation o, not any other operation.

Process Resilience

- Consensus in Faulty Systems with Arbitrary Failures: Practical Byzantine Fault Tolerance



- 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.

Process Resilience

- Consensus in Faulty Systems with Arbitrary Failures: Practical Byzantine Fault Tolerance

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.

Solution

To do this, we need to prevent two different operations with the same timestamp from getting committed, regardless of the view they're associated with. This is done by requiring 2k+1 commit certificates based on prepare certificates. We regenerate commit certificates for the new view to make sure no non-faulty server misses any operation. If a server has already executed an operation, it will ignore the duplicate certificate as long as it tracks its execution history.

Process Resilience

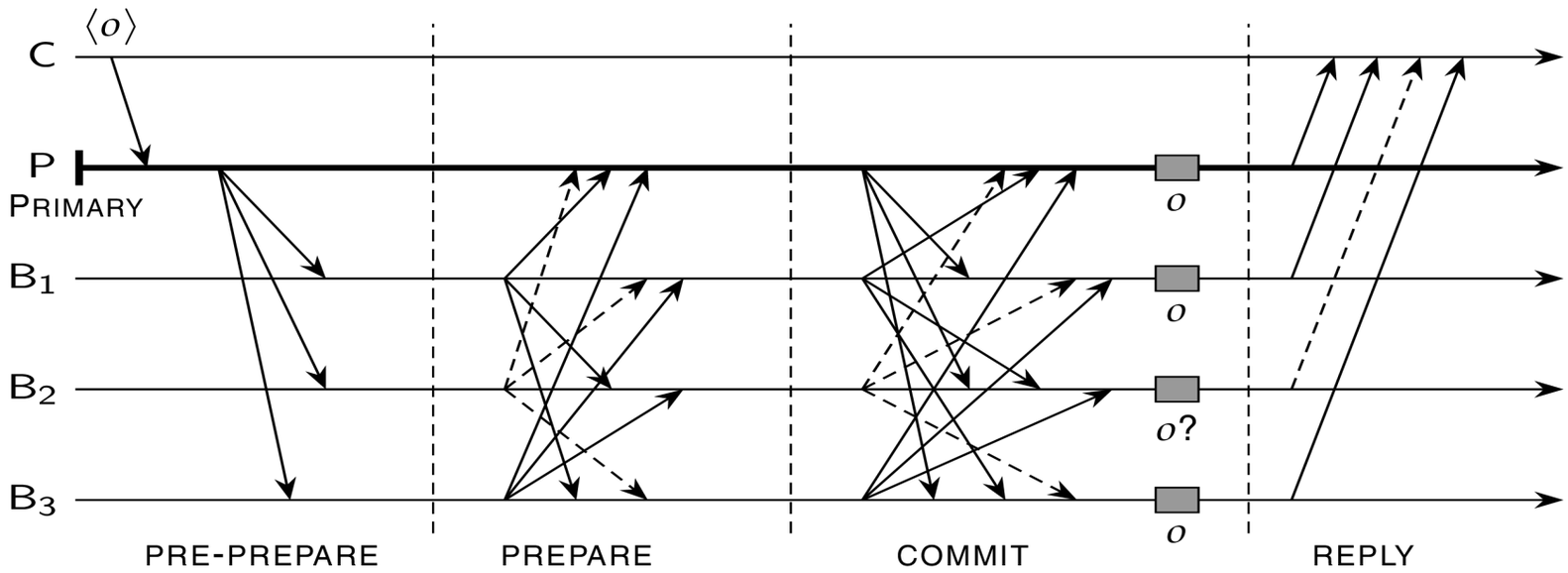
- Consensus in Faulty Systems with Arbitrary Failures: Practical Byzantine Fault Tolerance

Procedure

- When a backup server detects the failure, it broadcasts a signed view-change(v +1, P) message, with P being its set of prepare certificates. PBFT includes a deterministic function known to all backups that decides who will be the next primary. The new primary waits until it receives 2k +1 view-change messages, forming a view-change certificate (X) of prepare certificates. It then broadcasts new-view(v +1, X, O), where O consists of new pre-prepare messages based on the following:
- If there's a prepare certificate PC(t,v′,o) in X, it sends pre-prepare(t,v +1,o) in O, unless there's a more recent certificate.
- If there's no prepare certificate in X, it sends pre-prepare(t,v +1, none).
- 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*.

Process Resilience

- Consensus in Faulty Systems with Arbitrary Failures: Practical Byzantine Fault Tolerance



Procedure

- This means any outstanding operations from the previous view are moved to the new view, considering only the most recent view. The new primary sends out appropriate new pre-prepare messages based on what the backups had already committed to.
- Each backup checks O and X to ensure all operations are authentic and broadcasts prepare messages for all pre-prepare messages in O. This brings us back to the situation shown in Figure, but with one of the backups now acting as the primary and the old primary as a faulty backup.

Process Resilience

- Some Limitations on Realizing Fault Tolerance

When we group replicated processes together, it helps increase fault tolerance. Basically, it means our systems can handle failures better. But here's the catch: it can come at the cost of performance. In the examples we've seen before, processes in a fault-tolerant group might need to exchange many messages before they can make a decision. A good example of this is the Byzantine agreement protocol, which shows just how closely these processes need to work together.

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?

Process Resilience

- Some Limitations on Realizing Fault Tolerance: On Reaching Consensus

If a client can make decisions through voting, we can handle up to k out of 2k+1 processes giving wrong results. But let's assume these processes don't combine to mess things up. It gets difficult when we need the whole process group to agree, which is crucial in many cases.

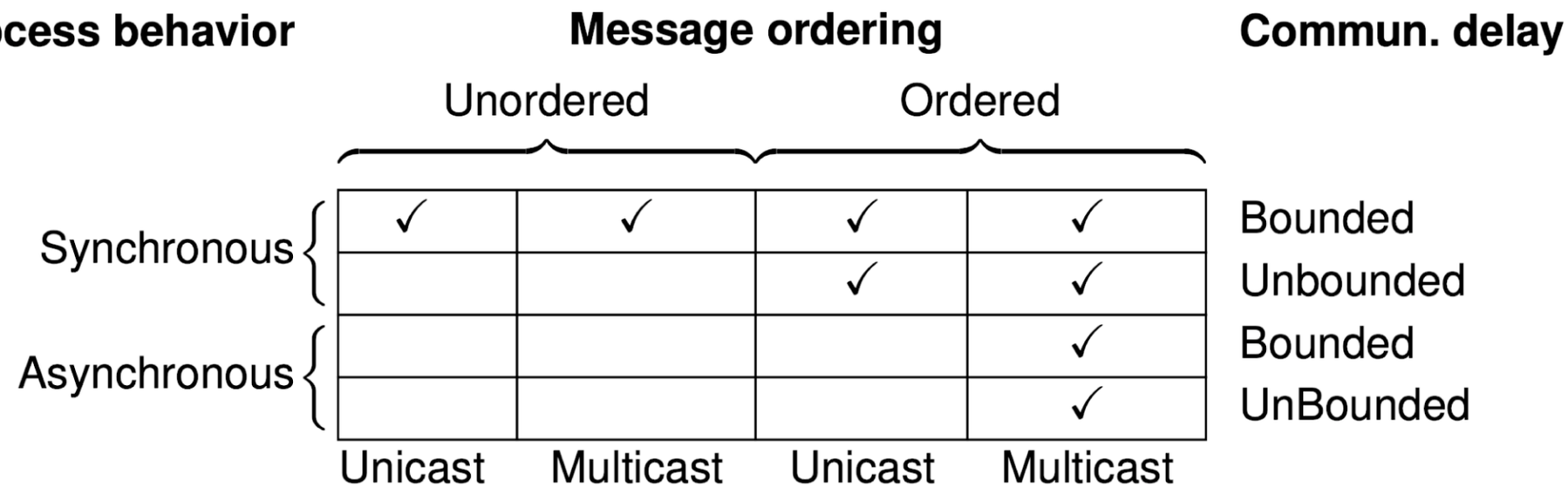
Formal requirements for consensus

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

Some situations where we need consensus include choosing a coordinator, deciding on a transaction, and dividing tasks among workers. When everything's working perfectly, reaching consensus is straightforward, but issues arise when things aren't perfect. The main goal of distributed consensus algorithms is to get all the non-faulty processes to agree on something within a limited number of steps. This gets complicated because different system assumptions require different solutions, if solutions even exist.

Process Resilience

- Some Limitations on Realizing Fault Tolerance: On Reaching Consensus



Cases

- **Synchronous vs. Asynchronous Systems**: A system is synchronous if processes operate in lock-step. This means there's a constant c such that if any process has taken c+1 steps, every other process has taken at least one step.
- **Communication Delay**: Delay is bounded if every message is delivered within a globally known maximum time.
- **Message Delivery Order**: This distinguishes between messages being delivered in the order they were sent and no such guarantee.
- **Message Transmission**: Unicasting vs. multicasting.
- Reaching Consensus in specific situations

Process Resilience

- Some Limitations on Realizing Fault Tolerance: Consistency, Availability, and Partitioning

  In the real world, the assumption that processes in a group can always communicate might be false. Messages can get lost, and networks can fail, causing group partitions.

  ### 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, like network failures

  ### 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.

Process Resilience

- 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 = a known crash

Process Resilience

- 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 Client-Server Communication

- RPC Semantics in the Presence of Failures

In client-server communication, use high-level tools like remote procedure calls (RPCs). The whole idea behind RPC is to make remote calls look and feel just like local ones. And for the most part, it works well—as long as everything is running smoothly on both ends.

What can go wrong? Failures in RPC systems

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.

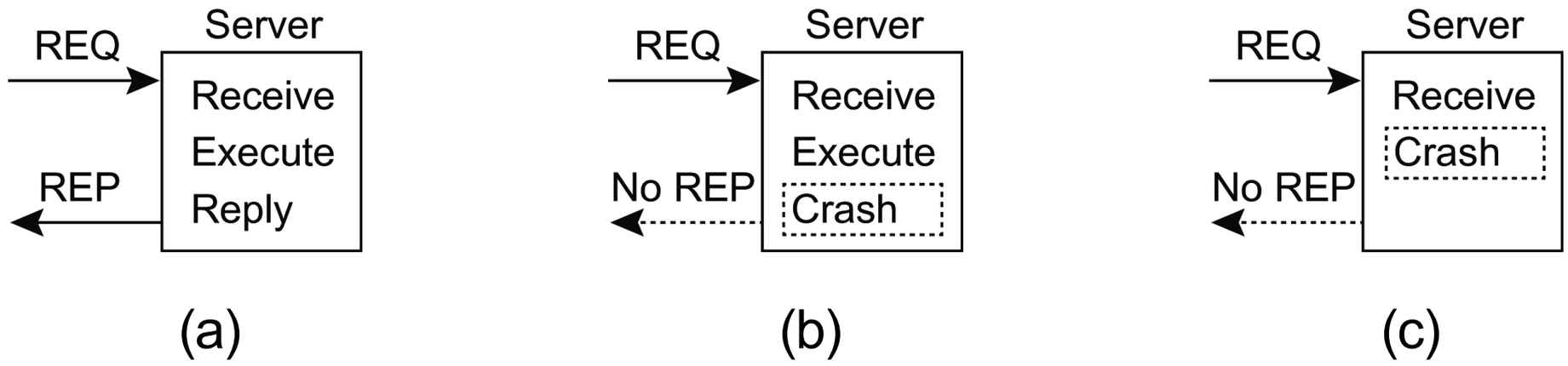
Two “easy” solutions

1: (cannot locate server): just report back to client

2: (request was lost): just resend message

Reliable Client-Server Communication

- RPC Semantics in the Presence of Failures: Server Crash



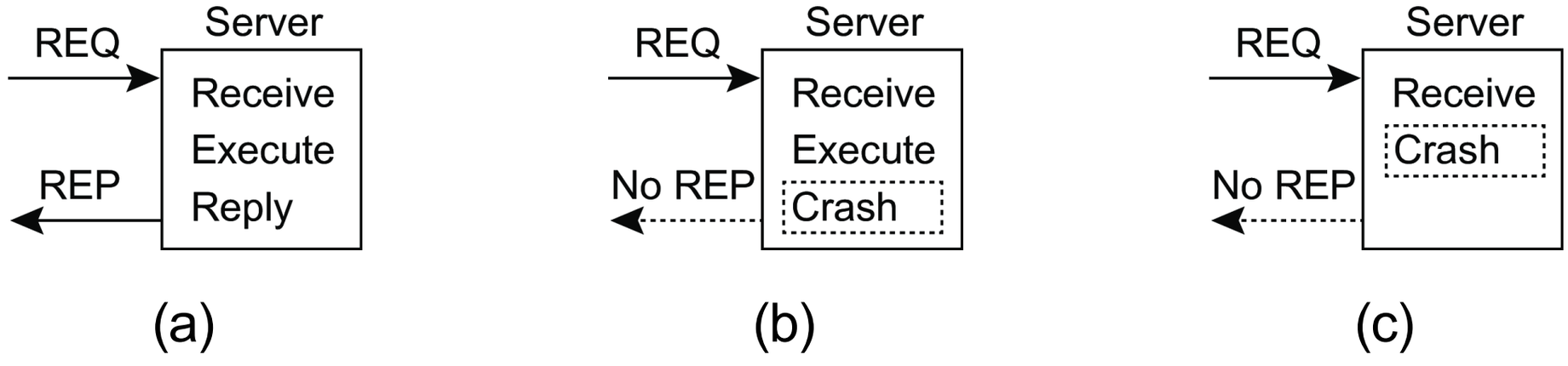
Crash cases

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

- (a): Traditionally, a request arrives, gets processed, and a reply is sent back to the client.
- (b): The server crashes after processing the request but before sending the reply.
- (c): The server might crash before even processing the request, so no reply is sent.

Reliable Client-Server Communication

- RPC Semantics in the Presence of Failures: Server Crash



Problem

The difficult part is how to handle these different situations. If the server crashes after processing the request but before replying, we need to report the failure to the client. But if the server crashes before processing, we can just resend the request. The problem is, the client's operating system can't tell which situation it's dealing with—it just knows that its timer has expired.

Two approaches:

- At-least-once-semantics: Keep trying until you get a reply, meaning the request is guaranteed to be processed at least once, but maybe more.
- At-most-once-semantics: Give up immediately and report a failure, ensuring the request is processed at most once, but possibly not at all.

Reliable Client-Server Communication

- RPC Semantics in the Presence of Failures: Server Crash

Problem

None of these are perfect. Ideally, we'd like exactly-once semantics, but that's generally impossible. Here's an example: Suppose the remote operation is processing a document to produce a PDF. The server sends a completion message when done, and the client gets an acknowledgment that the request was delivered. If the server crashes and then recovers, it announces its recovery to all clients. But the client doesn't know if its document processing request was completed.

Four strategies:

- Never reissue the request, risking that the document won't be processed.
- Always reissue the request, which might result in processing the document twice, causing extra work.
- Reissue the request only if it didn't get an acknowledgment, assuming the server crashed before receiving the request.
- Reissue the request only if it got an acknowledgment, assuming the server got the request but crashed before processing.

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* → *P* → *C*: The server sends the completion message, processes the document, and then crashes.
2. *M* → *C* → *P*: The server sends the completion message but crashes before the document is fully processed.
3. *P* → *M* → *C*: The server processes the document, sends the completion message, and then crashes.
4. *P* → *C* (→ *M*): The server processes the document but crashes before sending the completion message.
5. *C* (→ *P* → *M*): The server crashes before it can complete the document processing.
6. *C* (→ *M* → *P*): The server crashes before it can do anything at all.

Reliable Client-Server Communication

- RPC Semantics in the Presence of Failures

Reissue strategy
Always
Never
Only when ACKed
Only when not ACKed

Client

Strategy M → P		
MPC	**MC(P)**	**C(MP)**
DUP	OK	OK
OK	ZERO	ZERO
DUP	OK	ZERO
OK	ZERO	OK

Server

Strategy P → M		
PMC	**PC(M)**	**C(PM)**
DUP	DUP	OK
OK	OK	ZERO
DUP	OK	ZERO
OK	DUP	OK

Server

OK = Document processed once
DUP = Document processed twice
ZERO = Document not processed at all

All possible combinations

There's no combination of client and server strategies that works perfectly for all event sequences. The bottom line is that the client can never know if the server crashed just before or just after completing the document processing. server crashes completely change the nature of RPCs and clearly highlight the difference between single-processor and distributed systems. In a single-processor system, if the server crashes, the client crashes too, so there's no need for recovery. But in a distributed system, we can and should take action to handle these crashes.

Reliable Client-Server Communication

- RPC Semantics in the Presence of Failures: Lost Reply Messages

The real issue

Lost replies can be difficult to handle. The obvious solution is to use a timer set by the client's operating system. If no reply comes within a reasonable time, just resend the request. 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. But consider a request to a banking server to transfer money from one account to another. If the request is received and processed but the reply gets lost, the client won't know and will resend the request. The bank server will see this as a new request and process it again, transferring the money twice. Transferring money is not idempotent.

Reliable Client-Server Communication

- RPC Semantics in the Presence of Failures: Lost Reply Messages

Partial solution

Another approach is for the client to assign each request a sequence number. The server keeps track of the most recent sequence number from each client. This way, the server can tell if a request is a new one or a retransmission and avoid carrying out the same request twice. The server still needs to send a response back to the client, though. This method requires the server to keep track of each client's requests, and it's not always clear how long to maintain this record. An additional safeguard is to include a bit in the message header to distinguish initial requests from retransmissions. The idea is that it's always safe to perform an original request, but retransmissions might need more careful handling.

Reliable Client-Server Communication

- RPC Semantics in the Presence of Failures: Client Crash

Problem

What happens if a client sends a request to the server and then crashes before getting a reply? In this case, the server is still doing its work, but no one is waiting for the result. This remaining work is called an orphan computation. Orphan computations can cause several problems. First, they waste processing power. They can also lock files or tie up other valuable resources. Plus, if the client reboots and sends the request again, but then gets a reply from the orphan computation, things can get really confusing. This is basically about ensuring at-most-once semantics and restoring the client to its pre-crash state. One solution is checkpointing—saving the client's state just before sending a request and restoring it if needed.

Reliable Client-Server Communication

- RPC Semantics in the Presence of Failures: Client Crash

Solution

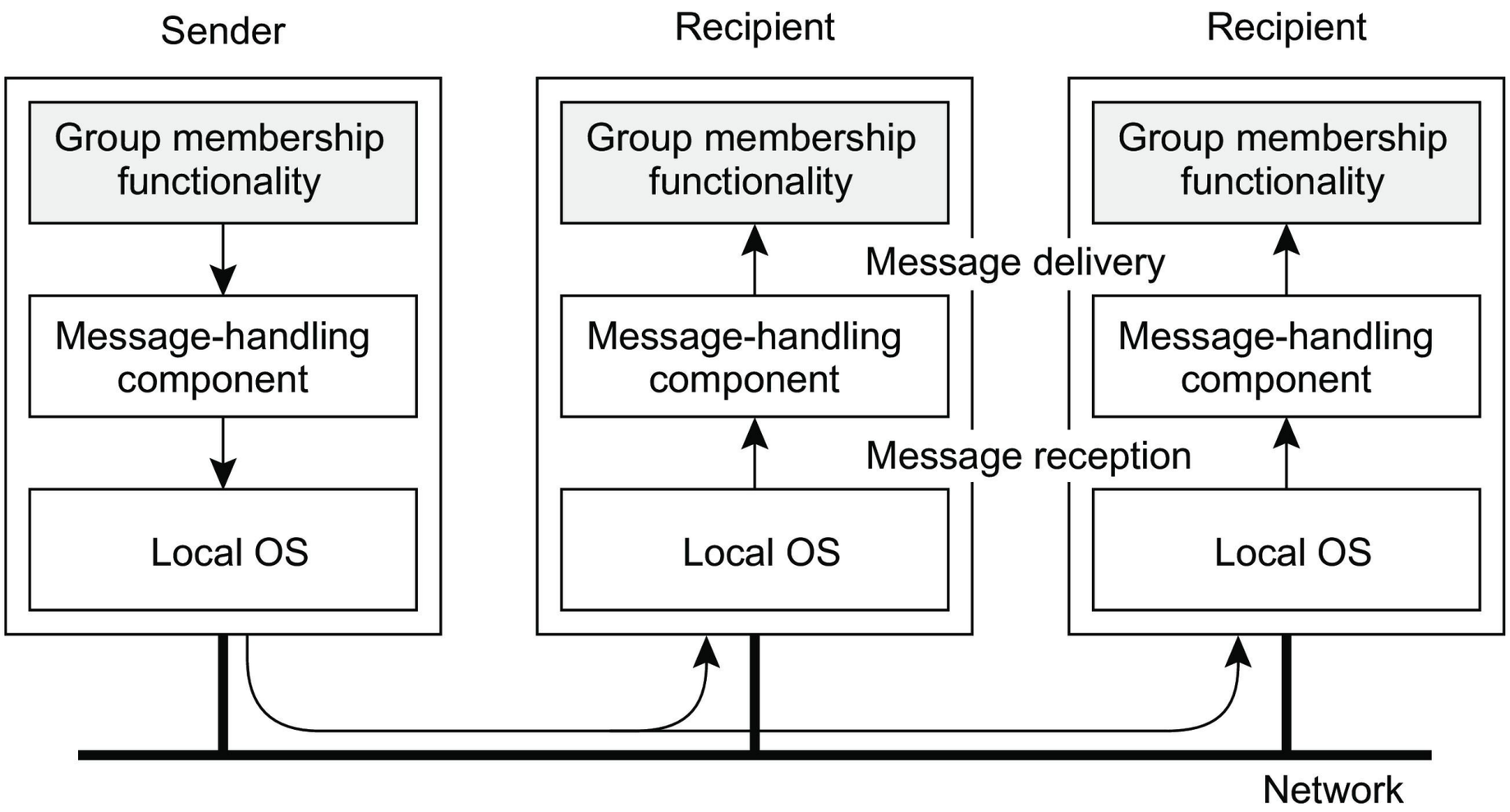
- Orphan Extermination: Before sending an RPC message, the client logs what it's about to do. If it crashes and reboots, it checks the log and kills the orphan. This approach has drawbacks, like the cost of writing to the log and the complexity of tracking down orphans and their descendants.
- Reincarnation: This solution divides time into numbered epochs. When a client recovers, it broadcasts a message declaring a new epoch, and all remote computations are killed. If the network is partitioned, some orphans might survive, but they can be detected by their outdated epoch numbers.
- Gentle Reincarnation: Similar to reincarnation, but less harsh. When an epoch broadcast is received, each machine checks for remote computations and tries to find their owners. Only if the owners can't be found are the computations killed.
- Expiration: Each RPC is given a set amount of time (T) to complete. If it needs more time, it must ask for an extension. After a client crash, waiting for T ensures all orphans are gone. The challenge is choosing a reasonable value for T.

Reliable Group Communication

- Introduction

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.



Reliable Group Communication

- Introduction

Intuition

Reliable group communication means that a message sent to a process group should reach each member of that group. We can separate the task of handling messages from the core operations of a group member, making it easier to manage message delivery. For example, a message sent to process P should be received and delivered to P's core component. Ensuring that messages from the same sender are delivered in the correct order is handled by a message-handling component. Reliable group communication can be more precisely defined by separating message reception and delivery. We need to distinguish between reliable communication when processes are faulty and when they operate correctly. For reliable communication in the presence of faults, a message should be received and delivered to all non-faulty group members.

Reliable Group Communication

- Introduction

Intuition

One difficult aspect is agreeing on the group's membership before delivering a message. If a sender wants a message delivered to group G, but the group changes to G′ by the time of delivery, we need to decide if the message can still be delivered. If we ignore the need for consensus on group membership, things get simpler. For example, if the sender has a list of recipients, it can use reliable protocols like TCP to send the message to each recipient one by one. If a recipient fails, the message can be resent later or ignored if the sender has left the group. Communication can be sped up by separating the sending of a request from receiving a response, as shown in message sequence charts.

Reliable Group Communication

- Introduction

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. If a sender meant for a message to be delivered to every member of group G, but at the time of delivery the group has changed to G′ (not equal to G), we need to decide whether that message should still be delivered.

Reliable Group Communication

- Introduction

Reliable communication, but assume nonfaulty processes

Most transport layers are good at reliable point-to-point communication but not so much for reliable group communication, namely, reliable multicasting: is a message received and delivered to each recipient, as intended by the sender.

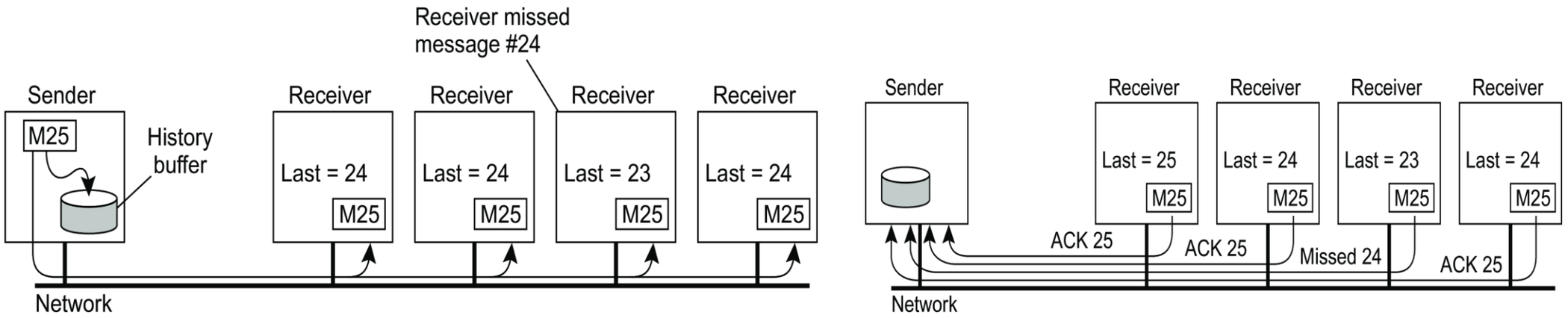
When you have a small group, this is a simple and practical solution. However, we often have to deal with systems where multicasting is unreliable—meaning a multicast message might get lost along the way and only some of the intended receivers get it.

Reliable Group Communication

- Introduction

Ways to achieve reliable group communication

1. The sender assigns a sequence number to each message it multicasts and stores the message in a local history buffer.
2. Assuming the receivers are known to the sender, the sender keeps the message in its history buffer until it gets an acknowledgement from each receiver.
3. If a receiver notices it's missing a message with a specific sequence number (because it has received messages with higher numbers), it sends a negative acknowledgement back to the sender, asking for a retransmission.



Distributed Commit

Problem

Making sure 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, committing a transaction at each site that's part of the transaction.

Distributed Commit

- Two-phase Commit Protocol (2PC)

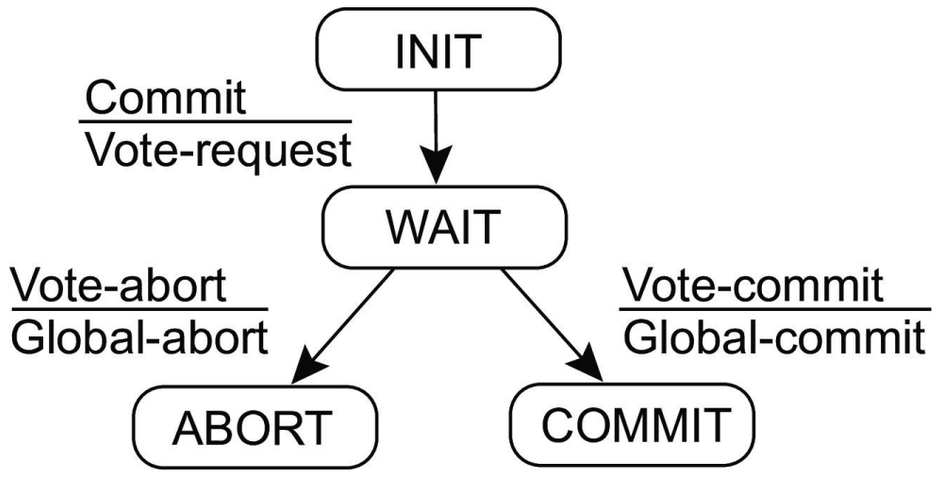
Essence

The client who initiated the computation acts as coordinator; processes required to commit are the participants. This is known as a one-phase commit protocol. The problem with this approach is that if one of the participants can't actually perform the operation, it has no way to inform the coordinator. For example, in the case of distributed transactions, a local commit might not be possible because it would violate concurrency control rules. In practice, we need more sophisticated methods. The most common one is the two-phase commit protocol.

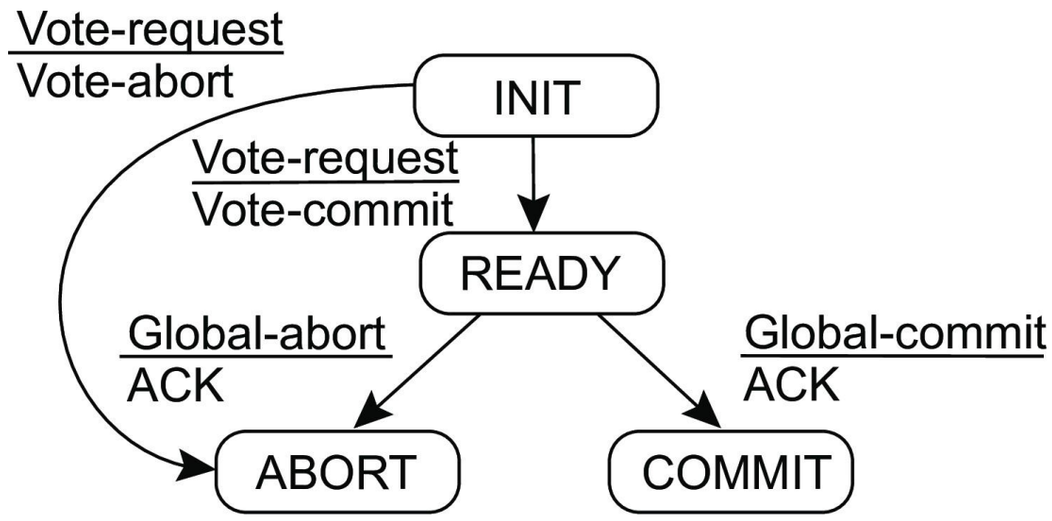
- Phase 1a: Coordinator sends VOTE-REQUEST to participants (also called a pre-write)
- Phase 1b: When participant receives VOTE-REQUEST it returns either VOTE-COMMIT or VOTE-ABORT to coordinator. If it sends VOTE-ABORT, it aborts its local computation
- Phase 2a: Coordinator collects all votes; if all 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.

Distributed Commit

- 2PC - Finite State Machines



Coordinator



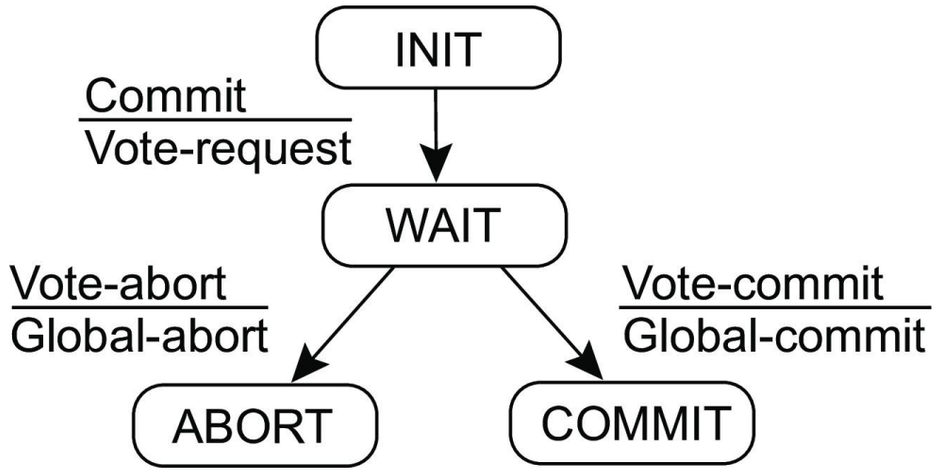
Participant

Essence

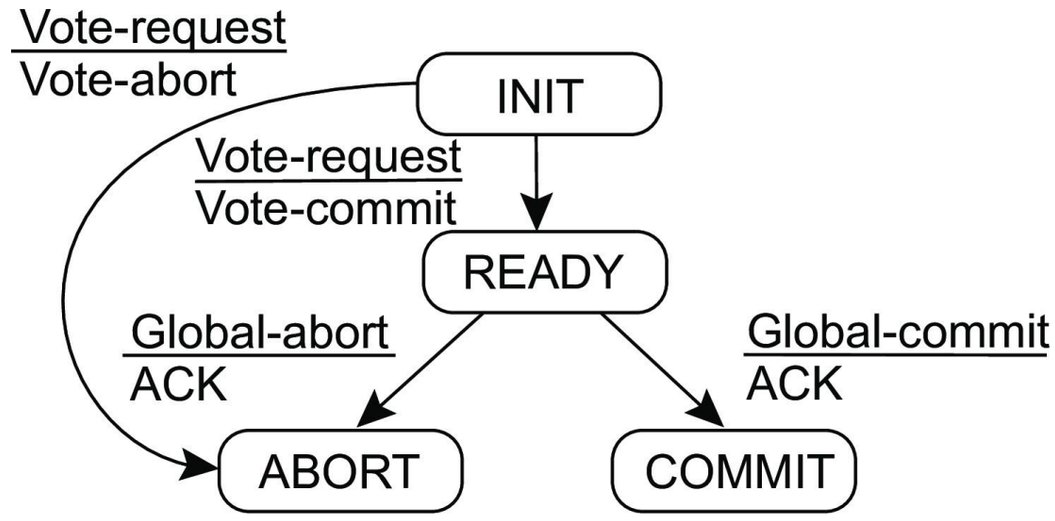
The first phase is the voting phase, which includes steps 1 and 2. The second phase is the decision phase, which includes steps 3 and 4. There are some issues when using the basic 2PC (two-phase commit) protocol in a system where failures can happen. First, both the coordinator and participants have states where they wait for incoming messages. This means the protocol can fail if a process crashes because other processes might be stuck waiting for a message that never arrives. To handle this, we use timeout mechanisms.

Distributed Commit

- 2PC - Finite State Machines



Coordinator



Participant

Essence

INIT State (Participant): The participant is waiting for a vote-request message from the coordinator. If it doesn't get this message in time, it decides to locally abort the transaction and sends a vote-abort message to the coordinator.

WAIT State (Coordinator): The coordinator is waiting for votes from all participants. If not all votes come in within a certain time, the coordinator decides to abort and sends a global-abort message to all participants.

READY State (Participant): The participant is waiting for the global vote from the coordinator. If this message doesn’t arrive in time, the participant can’t just abort the transaction. Instead, it needs to find out what the coordinator actually decided. The simplest solution is for the participant to block until the coordinator recovers.

Distributed Commit

- 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 ⇒ 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.

Distributed Commit

- 2PC – Failing Participant

To ensure that a process can actually recover after a crash, it needs to save its state to persistent storage:

- If a participant was in the INIT state, it can safely decide to abort the transaction when it recovers and inform the coordinator.
- If it crashed while in the COMMIT or ABORT state, it should recover to that state and resend its decision to the coordinator.
- The difficult part is if a participant crashes while in the READY state. In this case, when it recovers, it can't decide on its own whether to commit or abort. It needs to contact other participants to find out what to do next, similar to when it times out in the READY state.

The coordinator has two critical states to keep track of

1. WAIT State: When starting the 2PC protocol, the coordinator should record that it's entering the WAIT state. This way, it can resend the vote-request message to all participants after recovering.
2. Decision State: If the coordinator has made a decision in the second phase, it needs to record that decision so it can be resent after recovering.

Distributed Commit

- 2PC – Failing Participant

Alternative

A better solution than just waiting is to have participant P contact another participant Q to see if Q's current state can help P decide what to do. For example, if Q is in the COMMIT state, it means the coordinator must have sent a global-commit message to Q just before crashing, even though P didn't get it. So, P can safely decide to commit as well. If Q is in the ABORT state, then P can also safely abort. Now, let's say Q is still in the INIT state. This could happen if the coordinator sent a vote-request to all participants, but it reached P (which responded with a vote-commit) and not Q. This means the coordinator crashed while sending the vote-request. In this case, it's safe for both P and Q to move to the ABORT state.

Result

The toughest situation is when Q is also in the READY state, waiting for a response from the coordinator. If all participants are in the READY state, no one can make a decision. Even though everyone is willing to commit, they still need the coordinator's final decision. So, the protocol blocks until the coordinator recovers.

Distributed Commit

- 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 *Q*	**Action by** *P*
COMMIT	Make transition to *COMMIT*
ABORT	Make transition to *ABORT*
INIT	Make transition to *ABORT*
READY	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.

Distributed Commit

- 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

Recovery

- Introduction

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. You need to know in advance what errors might occur so you can correct them.
- Backward error recovery: Bring the system back into a previous error-free state. To do this, we record the system's state from time to time—this is called taking a checkpoint. If something goes wrong, we restore the system to one of these checkpoints.

Example: reliable communication

If a packet gets lost, the common approach is to have the sender retransmit it. This is backward recovery because we're trying to go back to the state where the packet was sent. This method is widely used because it's general and can be applied to any system or process.

Recovery

- Introduction

Backward recovery downsides:

- Performance Cost: Restoring a system or process to a previous state can be costly in terms of performance. For instance, recovering from a process crash or site failure usually involves a lot of work.
- No Guarantees: Since backward recovery is independent of the application, there's no guarantee that the same failure won't happen again. Sometimes, the application itself needs to be involved in the recovery process.
- Irreversible States: Some states can’t be rolled back. For example, if a malfunctioning ATM spits out $1,000 and someone takes it, we can’t just put the money back in the machine.

Checkpointing allows us to return to a previous correct state, but it's often costly and can slow down performance. So, many fault-tolerant systems combine checkpointing with message logging.

- Sender-Based Logging: After taking a checkpoint, a process logs its messages before sending them.
- Receiver-Based Logging: The receiving process logs the incoming message before delivering it to the application.

Recovery

- Introduction

Observation

When a receiving process crashes, we restore the most recent checkpoint and replay the logged messages. This way, we can restore a state beyond the most recent checkpoint without the full cost of checkpointing. If we only use checkpointing, processes go back to checkpointed states, but their behavior might change due to differences in communication times. With message logging, we can replay events exactly as they happened, making it easier to maintain consistency. For example, if a failure was caused by a user's erroneous input, with just checkpointing, we'd need a checkpoint right before the user's input to recover correctly. With message logging, we can use an older checkpoint and replay events up to the point of the user's input. Combining fewer checkpoints with message logging is generally more efficient than taking many checkpoints.

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

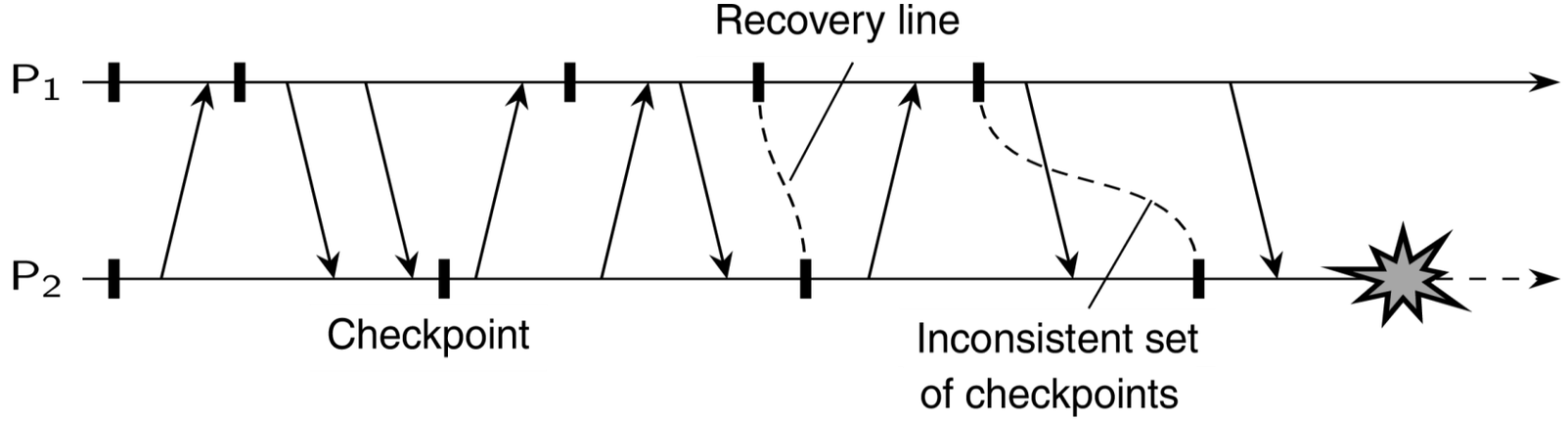
Recovery

- Checkpointing

Requirement

In a fault-tolerant distributed system, backward error recovery means that the system needs to regularly save its state. Specifically, we need to record a consistent global state, also known as a distributed snapshot. In a distributed snapshot, if process P has recorded receiving a message, then there must be a process Q that recorded sending that message. After all, the message had to come from somewhere.

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



Recovery line

To recover after a process or system failure, we need to reconstruct a consistent global state from the local states saved by each process. Ideally, we recover to the most recent distributed snapshot, also called a recovery line. This recovery line is the most recent consistent collection of checkpoints.

Recovery

- Checkpointing: Coordinated Checkpointing

Essence

In coordinated checkpointing, all processes synchronize to save their state to local storage at the same time. The big advantage of this method is that the saved state is automatically globally consistent.

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

Recovery

- Checkpointing: Coordinated Checkpointing

Observation

This approach ensures a globally consistent state because no incoming messages are included in the checkpoint. Any messages that arrive after the checkpoint request are not part of the local checkpoint, and outgoing messages are queued until the checkpoint-done message is received. To improve this algorithm, we can send the checkpoint request only to processes that depend on the recovery of the coordinator and ignore the rest. A process depends on the coordinator if it has received a message directly or indirectly related to a message the coordinator sent since the last checkpoint. This concept is called an incremental snapshot. For an incremental snapshot, the coordinator sends a checkpoint request only to the processes it communicated with since the last checkpoint. When a process P gets this request, it forwards it to all processes it has sent messages to since the last checkpoint, and so on. Each process forwards the request only once. After identifying all relevant processes, a second multicast is used to trigger the actual checkpointing and let the processes resume their activities.

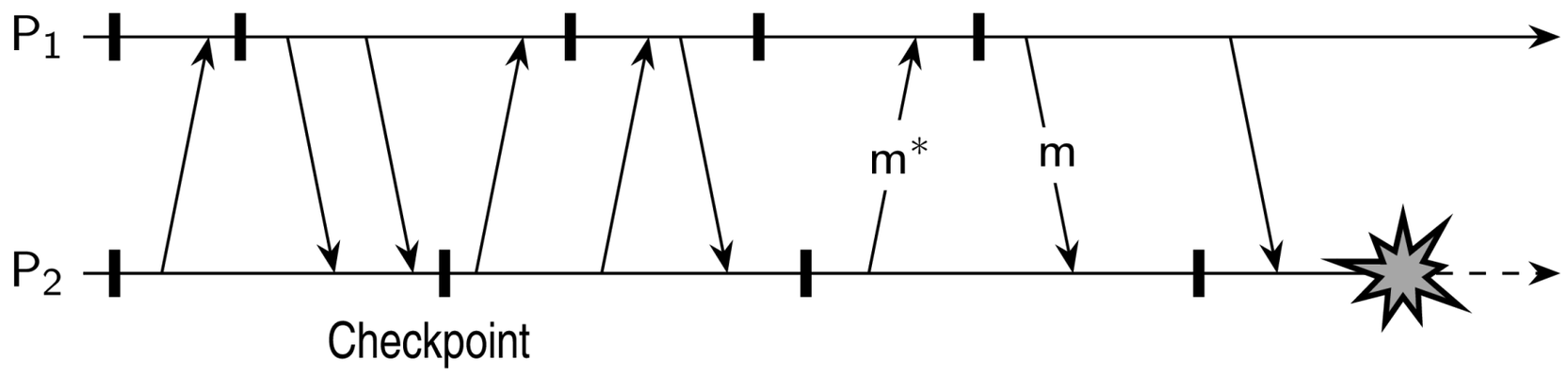
Recovery

- Checkpointing: Independent Checkpointing

Observation

What happens when each process saves its local state independently from time to time, without any coordination. If checkpointing is done at the "wrong" instants, the recovery line may lie at system startup time. We have a so-called cascaded rollback.

To find a recovery line, each process needs to roll back to its most recently saved state. If these states don't form a consistent snapshot, we have to keep rolling back further. This process can lead to a domino effect.

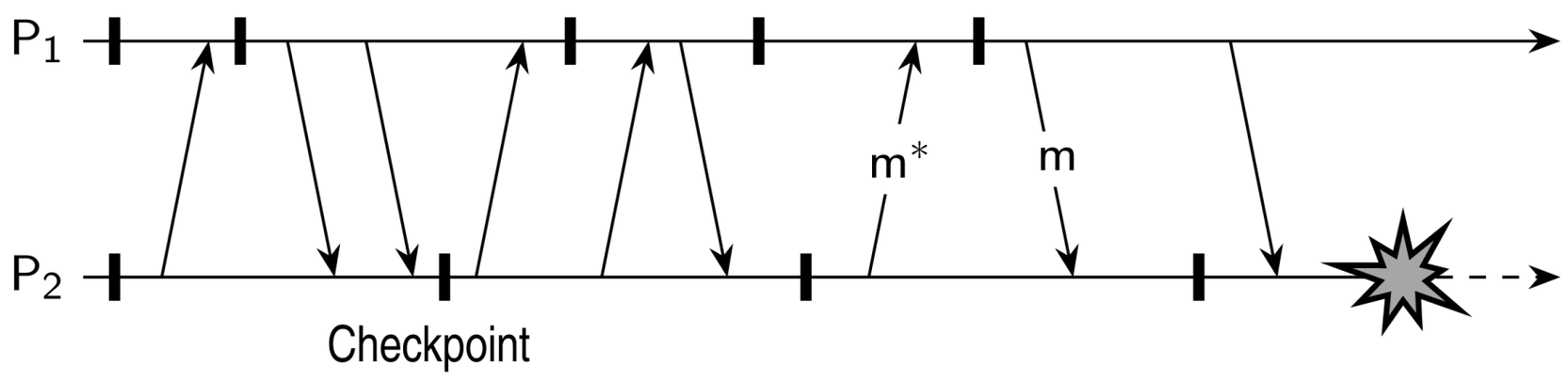


Recovery

- Checkpointing: Independent Checkpointing

Observation

If process P2 crashes, we need to restore its state to its last saved checkpoint. As a result, process P1 also needs to roll back. Unfortunately, these two most recent local states don’t form a consistent global state: P2's saved state indicates it received a message, but no other process shows it sent that message. So, P2 has to roll back even further. But, the next state to which P2 rolls back might still not be usable as part of a distributed snapshot. In this case, P1 might have recorded the receipt of another message, but there's no record of that message being sent. This means P1 also needs to roll back further. In this example, the recovery line ends up being the initial state of the system.



Recovery

- Checkpointing: 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 includes the pair (i, m) with the message
- 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 $CP_j(n-1)$.

Recovery

- 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.

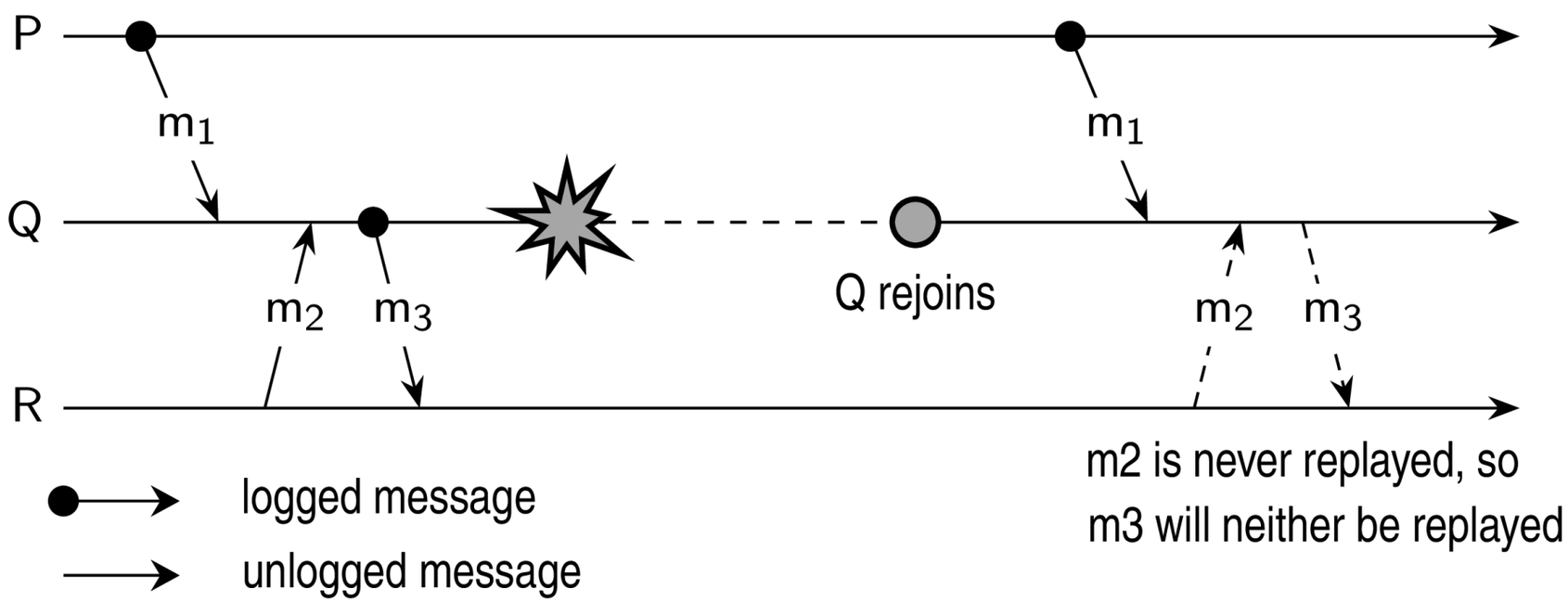
Recovery

- Message Logging

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.



Thank You