Fault Tolerance

Fault Tolerance

- Fault tolerance is the ability of a distributed system to provide its services even in the presence of faults.
- A distributed system should be able to recover automatically from partial failures without seriously affecting availability and performance.

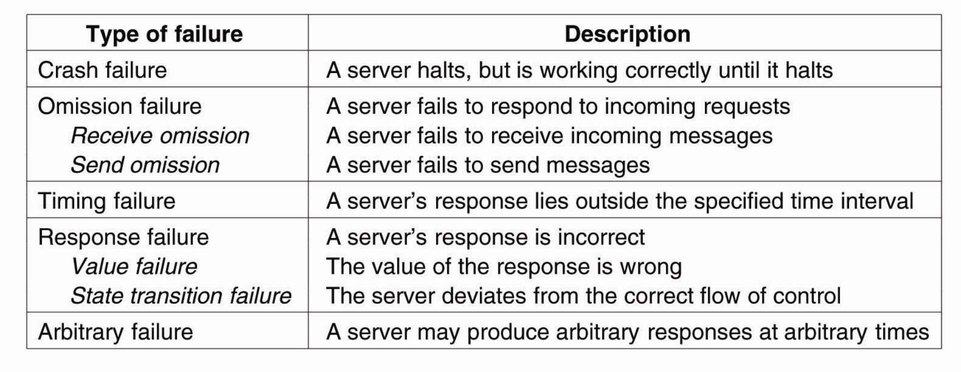
Fault Tolerance: Basic Concepts (1)

- Being fault tolerant is strongly related to what are called dependable systems. Dependability implies the following:
  - Availability: The percentage of the time the system is operational.
  - Reliability: The time interval for which the system is operational.
  - Safety
  - Maintainability
- **In-class Exercise**: How is is availability different from reliability? How about a system that goes down for 1 millisecond every hour? How about a system that never goes down but has to be shut down two weeks every year?

Fault Tolerance: Basic Concepts (2)

- A system is said to fail when it cannot meet its promises. For example, if it cannot provide a service (or only provide it partially).
- An error is a part of the system's state that may lead to failure. The cause of an error is called a fault.
- Types of faults:
  - Transient: Occurs once and then disappears. If a operation is repeated, the fault goes away.
  - Intermittent: Occurs, then vanishes of its own accord, then reappears and so on.
  - Permanent: Continues to exist until the faulty component is replaced.
- **In-class Exercise**: Give examples of each type of fault for your project!

Failure Models (1)



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

Failure Models (2)

- Fail-stop: failures that can be reliably detected
- Fail-noisy: failures that are eventually detected
- Fail-silent: we cannot detect between a crash failure or omission failure
- Fail-safe: failures that do no harm
- Fail-arbitrary or Byzantine Failures:
  - Arbitrary failures where a server is producing output that it should never have produced, but which cannot be detected as being incorrect.
  - A faulty server may even be working with other servers to produce intentionally wrong answers!

Reliable Client-Server Communication

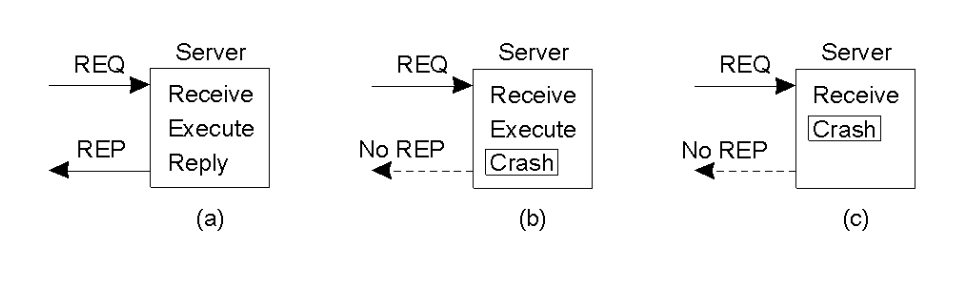
RMI semantics in the presence of failures.

- The client is unable to locate the server.
- The request message from the client to the server is lost.
- The server crashes after receiving a request.
- The reply message from the server to the client is lost.
- The client crashes after sending a request.

RPC Semantics in the Presence of Failures

- RPC failures
  - Client cannot locate server: Raise an exception or send a signal to client leading to loss in transparency
  - Lost request messages: Start a timer when sending a request. If timer expires before a reply is received, send the request again. Server would need to detect duplicate requests
  - Server crashes: Server crashes before or after executing the request is indistinguishable from the client side...
- Possible RPC semantics
  - Exactly once semantics
  - At least once semantics
  - At most once semantics
  - Guarantee nothing semantics

Server Crash (1)



- A server in client-server communication
  - (a) Normal case
  - (b) Crash after execution
  - (c) Crash before execution

Server Crash (2)

- Server: Prints text on receiving request from client and sends message to client after text is printed.
  - Send a completion message just before it actually tells the printer to do its work
  - Or after the text has been printed
- Client: After a server crashes and recovers.
  - Never to reissue a request.
  - Always reissue a request.
  - Reissue a request only if it did not receive an acknowledgment of its request being delivered to the server.
  - Reissue a request only if it has not received an acknowledgment of its print request.

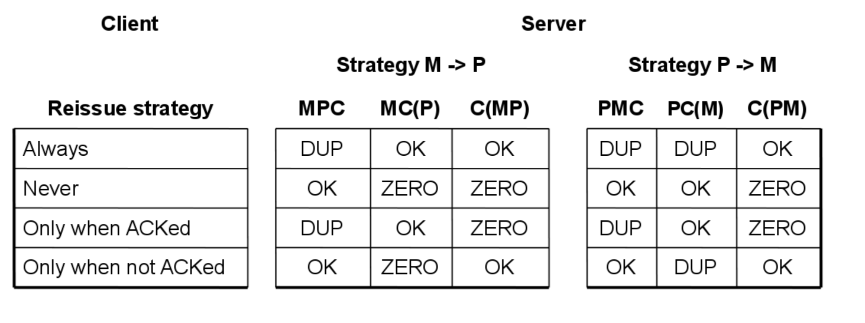
Server Crash (3)

- Three events that can happen at the server:
  - Send the completion message (M)
  - Print the text (P)
  - Crash (C)
- These events can occur in six different orderings:

Server Crash (4)

1. $M \rightarrow P \rightarrow C$: A crash occurs after sending the completion message and printing the text.
2. $M \rightarrow C(\rightarrow P)$: A crash happens after sending the completion message, but before the text could be printed.
3. $P \rightarrow M \rightarrow C$: A crash occurs after sending the completion message and printing the text.
4. $P \rightarrow C(\rightarrow M)$: The text printed, after which a crash occurs before the completion message could be sent.
5. $C(\rightarrow P \rightarrow M)$: A crash happens before the server could do anything.
6. $C(\rightarrow M \rightarrow P)$: A crash happens before the server could do anything.

Server Crash (5)



Client	Server					
	Strategy M -> P			Strategy P -> M		
Reissue strategy	MPC	MC(P)	C(MP)	PMC	PC(M)	C(PM)
Always	DUP	OK	OK	DUP	DUP	OK
Never	OK	ZERO	ZERO	OK	OK	ZERO
Only when ACKed	DUP	OK	ZERO	DUP	OK	ZERO
Only when not ACKed	OK	ZERO	OK	OK	DUP	OK

- *M*: send the completion message, *P*: print the text, *C*: server crash
- OK (text is printed once), DUP (text is printed twice), ZERO (text is not printed at all)

]

RPC Semantics in the Presence of Failures (2)

- Lost Reply Messages. Set a timer on client. If it expires without a reply, then send the request again. If requests are idempotent, then they can be repeated again without ill-effects
- Client Crashes. Creates orphans. An orphan is an active computation on the server for which there is no client waiting. How to deal with orphans:
  - Extermination. Client logs each request in a file before sending it. After a reboot the file is checked and the orphan is explicitly killed off. Expensive, cannot locate grand-orphans etc.
  - Reincarnation. Divide time into sequentially numbered epochs. When a client reboots, it broadcasts a message declaring a new epoch. This allows servers to terminate orphan computations.
  - Gentle Reincarnation. A server tries to locate the owner of orphans before killing the computation.
  - Expiration. Each RPC is given a quantum of time to finish its job. If it cannot finish, then it asks for another quantum. After a crash, a client need only wait for a quantum to make sure all orphans are gone.

Idempotent Operations

- An idempotent operation is one that can be repeated as often as necessary without any harm being done. E.g. reading a block from a file.
- In general, try to make RPC/RMI methods be idempotent if possible. If not, it can be dealt with in a couple of ways.
  - Use a sequence number with each request so server can detect duplicates. But now the server needs to keep state for each client....
  - Have a bit in the message to distinguish between original and duplicate transmission

Reliable Group Communication

- Reliable multicasting guarantees that messages are delivered to all members in a process group. Reliable multicasting turns out to be tricky.
- Basic reliable multicasting schemes
- Scalable reliable multicasting
  - Non-hierarchical feedback control
  - Hierarchical feedback control
- Atomic multicasting using Virtual Synchrony

Basic Reliable Multicasting Schemes (1)

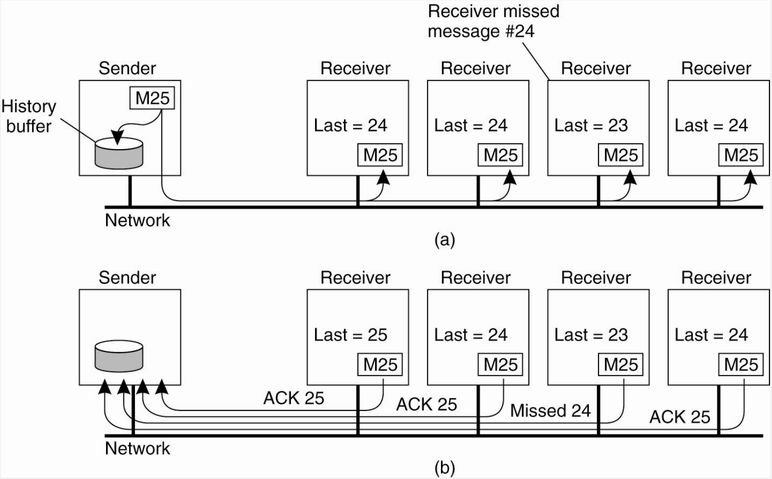
- Reliable point-to-point channels are available but reliable communication to a group of processes is rarely built-in to the transport layer. For example, multicasting uses datagrams, which are not reliable.
- Few processes: Set up reliable point-to-point channels. Inefficient for more processes.
- Issues in reliable multicasting:
  - What does reliable multicasting mean?
  - What if a process joins during the communication?
  - What happens if a sending process crashes during communication?
  - How to reach agreement on what does the group look like?

Basic Reliable Multicasting Schemes (2)

Assume that processes do not fail and join or leave the group during communication so group membership is known.

- Sending process assigns a sequence number to each message it multicasts. Messages are received in the order which they were sent.
- Each multicast message is kept in a history buffer at the sender. Assuming that the sender knows the receivers, the sender simply keeps the message until all receivers have returned the acknowledgment (`Ack`).
- Sender retransmits on a negative `Ack` or on timeout before all `Ack`s were received. `Ack`s can be piggy-backed. Retransmissions can be done with point-to-point communication.

Basic Reliable Multicasting Schemes (3)

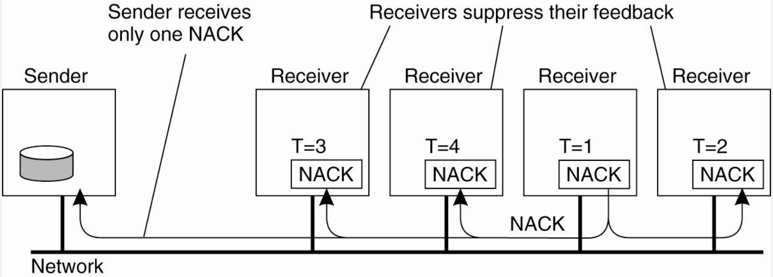


- A simple solution to reliable multicasting when all receivers are known and are assumed not to fail.
- (a) Message transmission. (b) Reporting feedback.

Scalability in Reliable Multicasting

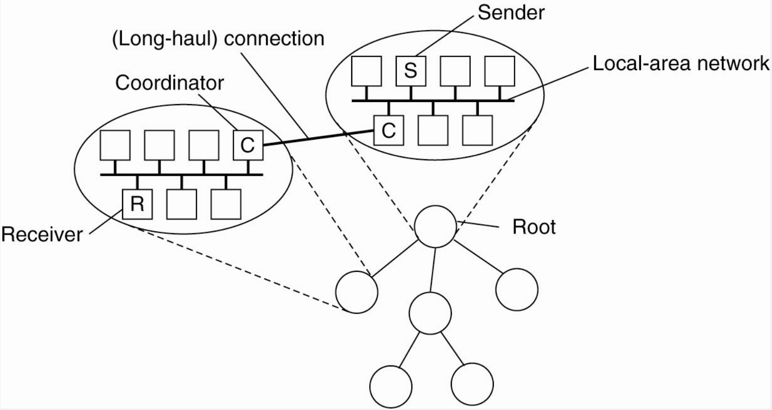
- Negative acknowledgments: A receiver returns feedback only if it is missing a message. This improves scalability by cutting down on the number of messages. However this forces the sender to keep a message in its buffer forever (so we need to use timeouts for the buffer)
- Nonhierarchical feedback control: Feedback suppression via multicasting of negative feedback.
- Hierarchical feedback control: Use subgroups and coordinators in each subgroup.

Nonhierarchical Feedback Control



- Several receivers have scheduled a request for retransmission with random delays, but the first retransmission request leads to the suppression of others.

Hierarchical Feedback Control

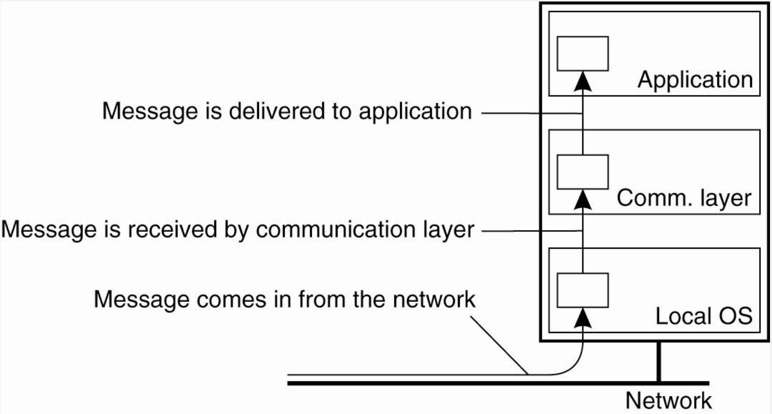


- The essence of hierarchical reliable multicasting. Each local coordinator forwards the message to its children and later handles retransmission requests.

Reliable Atomic Multicast

- The atomic multicast setup to achieve reliable multicasting in the presence of process failures requires the following conditions:
  - A message is delivered to all processes or to none at all.
  - All messages are delivered in the same order to all processes.
- For example, this solves the problem of a replicated database on top of a distributed system.
  - Atomic multicasting ensures that non-faulty processes maintain a consistent view of the database, and forces reconciliation when a replica recovers and rejoins the group.

Virtual Synchrony (1)


Application
Message is delivered to application
Comm. layer
Message is received by communication layer
Message comes in from the network
Local OS
Network

- The logical organization of a distributed system to distinguish between message receipt and message delivery.

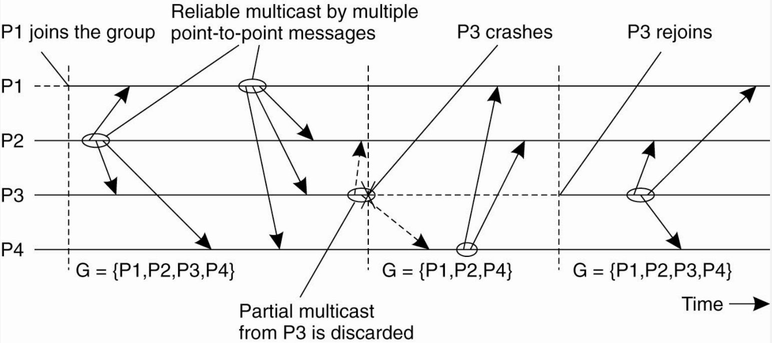
Virtual Synchrony (2)

- A multicast message m is uniquely associated with a list of processes to which it should be delivered. This delivery list corresponds to a group view. Each process on the list has the same view.
- A view change takes place by multicasting a message vc announcing the joining or leaving of a process.
- Suppose m and vc are simultaneously in transit. We need to guarantee that m is either delivered to all processes in the group view G before each of them is delivered message vc, or m is not delivered at all.
- Note that m being not delivered is because the sender of m crashed.

Virtual Synchrony (3)

- A reliable multicast is said to be virtually synchronous if it has the following properties:
  - A message multicast to group view G is delivered to each non-faulty process in G.
  - If a sender of the message crashes during the multicast, the message may either be delivered to all processes, or ignored by each of them.
- The principle is that all multicasts take place between view changes.
- All multicasts that are in transit when a view change takes place are completed before the view change comes into effect.

Virtual Synchrony (4)



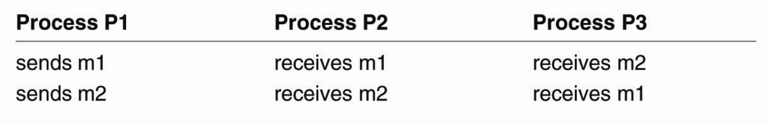
- The principle of virtual synchronous multicast.

Message Ordering in Multicasting (1)

Virtual synchrony allows us to think about multicasts as taking place in epochs. But we can have several possible orderings of the multicasts:

- Unordered multicasts
- FIFO-ordered multicasts
- Causally-ordered multicasts (requires vector timestamps)
- Totally-ordered multicasts

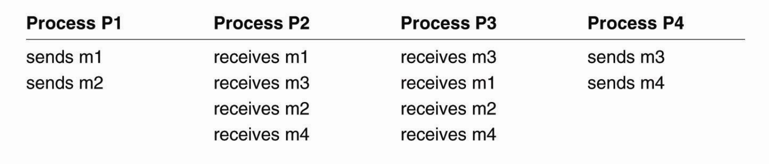
Message Ordering in Multicasting (2)



Process P1	Process P2	Process P3
sends m1	receives m1	receives m2
sends m2	receives m2	receives m1

- Three communicating processes in the same group. The ordering of events per process is shown along the vertical axis. This shows unordered multicasts.

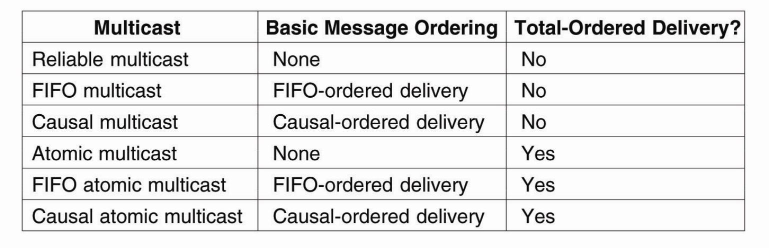
Message Ordering in Multicasting (3)



Process P1	Process P2	Process P3	Process P4
sends m1	receives m1	receives m3	sends m3
sends m2	receives m3	receives m1	sends m4
	receives m2	receives m2	
	receives m4	receives m4	

- Four processes in the same group with two different senders, and a possible delivery order of messages under FIFO-ordered multicasting.

Message Ordering in Multicasting (4)



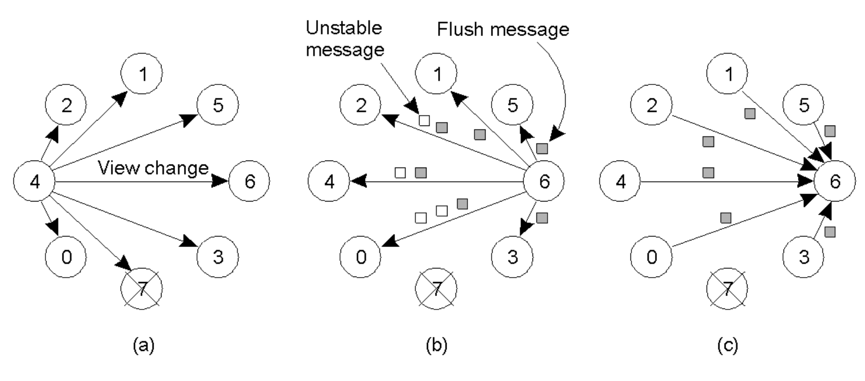
Multicast	Basic Message Ordering	Total-Ordered Delivery?
Reliable multicast	None	No
FIFO multicast	FIFO-ordered delivery	No
Causal multicast	Causal-ordered delivery	No
Atomic multicast	None	Yes
FIFO atomic multicast	FIFO-ordered delivery	Yes
Causal atomic multicast	Causal-ordered delivery	Yes

- Six different versions of virtually synchronous reliable multicasting.

Implementing Virtual Synchrony (1)

- Assume that we have reliable point to point communication (e.g. TCP) and that messages from the same source are received in the same order as sent.
- Every process in G keeps message m until it knows for sure that all member in G have received it. If m has been received by all members in G, then m is said to be stable. Only stable messages are allowed to be delivered.
- To ensure stability, it is sufficient to pick an arbitrary process in G and request it to send m to all other processes. That arbitrary process can be the coordinator.
- Assumes that no process crashes during a view change (although it can be generalized to handle that as well).

Implementing Virtual Synchrony (2)



- (a) Process 4 notices that process 7 has crashed, sends a view change.
- (b) Process 6 sends out all its unstable messages, followed by a flush message.
- (c) Process 6 installs the new view when it has received a flush message from everyone else.

Failure Masking by Redundancy

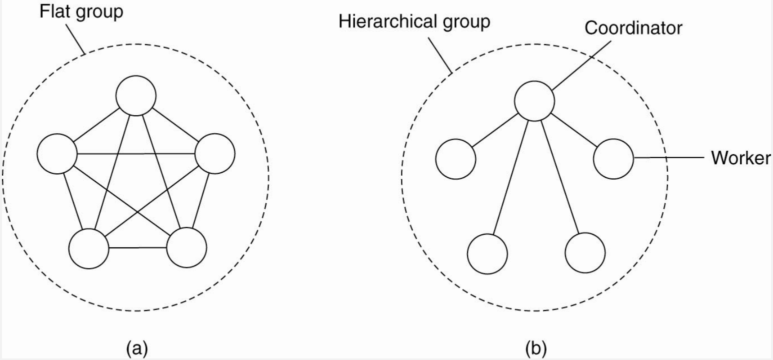
- Information Redundancy. For example, adding extra bits (like in Hamming Codes, see the book Coding and Information Theory) to allow recovery from garbled bits.
- Time Redundancy. Repeat actions if need be.
- Physical Redundancy. Extra equipment or processes are added to make the system tolerate loss of some components.

Process Resilience

Achieved by replicating processes into groups.

- How to design fault-tolerant groups?
- How to reach an agreement within a group when some members cannot be trusted to give correct answers?

Flat Groups Versus Hierarchical Groups



Failure Masking Via Replication

- *Primary-backup protocol*. A primary coordinates all write operations. If it fails, then the others hold an election to replace the primary.
- *Replicated-write protocols*. Active replication as well as quorum based protocols. Corresponds to a flat group.
- A system is said to be *k* fault tolerant if it can survive faults in k components and still meet its specifications.
  - For *fail-silent* components, $k + 1$ are enough to be k fault tolerant.
  - For *Byzantine failures*, at least $2k + 1$ extra components are needed to achieve k fault tolerance.
  - Requires atomic multicasting: all requests arrive at all servers in same order. This can be relaxed to just be for write operations.

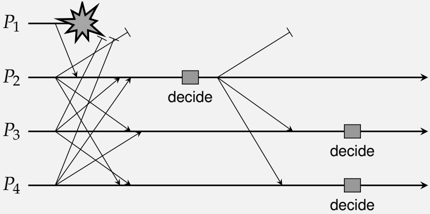
Consensus in Faulty Systems

The model: a very large collection of clients send commands to a group of processes that jointly behave as a single, highly robust process. To make this work, we need to make an important assumption:
In a fault-tolerant process group, each nonfaulty process executes the same commands, in the same order, as every other nonfaulty process.

- Flooding Consensus (for fail-stop failures)
- Paxos Consensus (for fail-noisy failures)
- Byzantine Agreement (for arbitrary failures)

Flooding Consensus

- A client contacts a group member ($P_1, \ldots, P_n$) requesting it to execute a command. Every group member maintains a list of commands: some from clients and some from other group members.
- In each round, a process P$_i$ sends its list of proposed commands it has seen to every other process. At the end of a round, each process merges all received proposed commands into a new list, from which it deterministically selects a new command to execute (if possible)
- A process will move on to the next round without having made a decision, only when it detects that another process has failed.

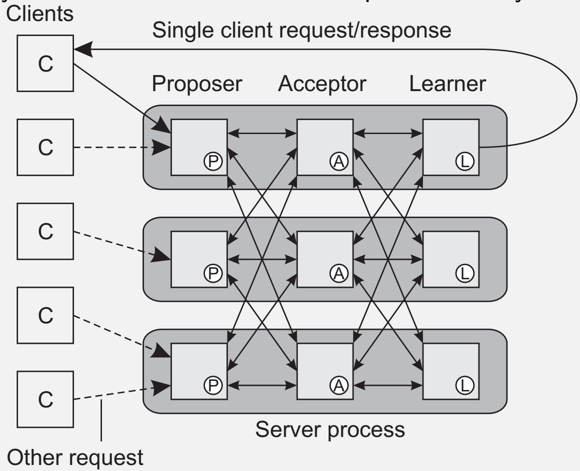


Example: Realistic Consensus: Paxos

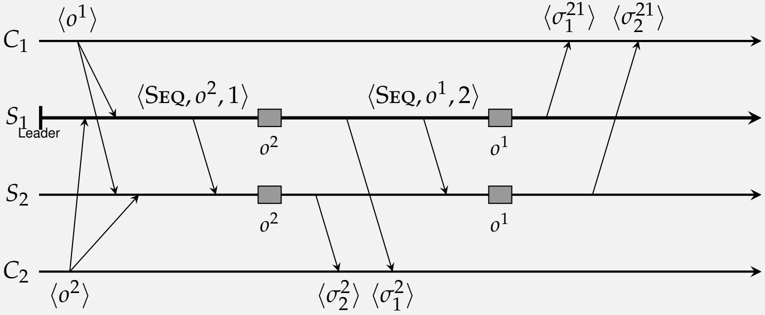
- **Assumptions (rather weak ones, and realistic)**
  - A partially synchronous system (in fact, it may even be asynchronous).
  - Communication between processes may be unreliable: messages may be lost, duplicated, or reordered.
  - Corrupted message can be detected (and thus subsequently ignored).
  - All operations are deterministic: once an execution is started, it is known exactly what it will do.
  - Processes may exhibit crash failures, but not arbitrary failures.
  - Processes do not collude.
- Understanding Paxos
  - We will build up Paxos from scratch to understand where many consensus algorithms actually come from.

Paxos Organization

Paxos protocol is named after a fictional legislative consensus system used on the Paxos island in Greece, where Lamport wrote that the parliament had to function "even though legislators continually wandered in and out of the parliamentary Chamber."



Two Server Situation


$\langle o^1 \rangle$
$\langle \sigma_1^{21} \rangle$
$\langle \sigma_2^{21} \rangle$
C_1
$\langle \textsc{Seq}, o^2, 1 \rangle$
$\langle \textsc{Seq}, o^1, 2 \rangle$
S_1
Leader
o^2
o^1
S_2
o^2
o^1
C_2
$\langle o^2 \rangle$
$\langle \sigma_2^2 \rangle$
$\langle \sigma_1^2 \rangle$

Subscripts designate processes, and superscripts designate operations and states.

Paxos: basics

- The leading proposer receives requests from clients, one at a time. A nonleading proposer forwards any client request to the leader.
- The leading proposer sends its proposal to all acceptors, telling each to request the operation.
- Each acceptor will subsequently broadcast a learn message.
- If a leader receives the same learn message from a majority of acceptors, it knows that consensus has been reached on which operation to execute, and will execute it.

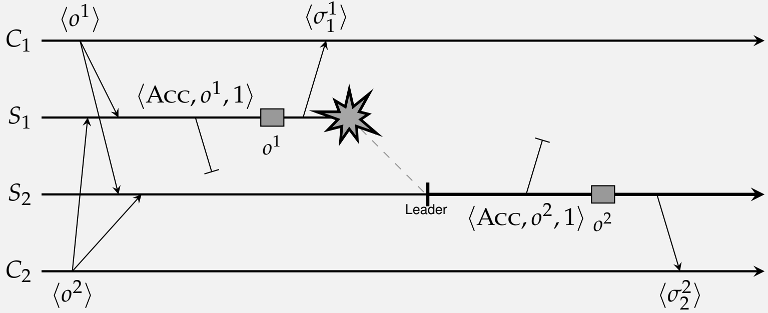
Issues:

- Not only do the servers need to reach consensus on which operation to execute, we also need to make sure that each of them actually executes it.
- Failing leader.

Handling lost messages

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

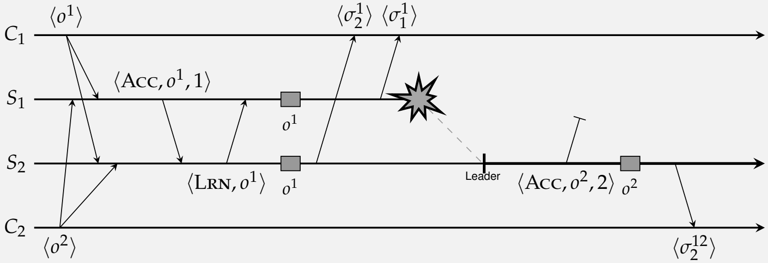
Two servers and one crash: problem


C_1
S_1
S_2
C_2
$\langle o^1 \rangle$
$\langle \sigma_1^1 \rangle$
$\langle \text{Acc}, o^1, 1 \rangle$
o^1
Leader
$\langle \text{Acc}, o^2, 1 \rangle$
o^2
$\langle o^2 \rangle$
$\langle \sigma_2^2 \rangle$

Problem:

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

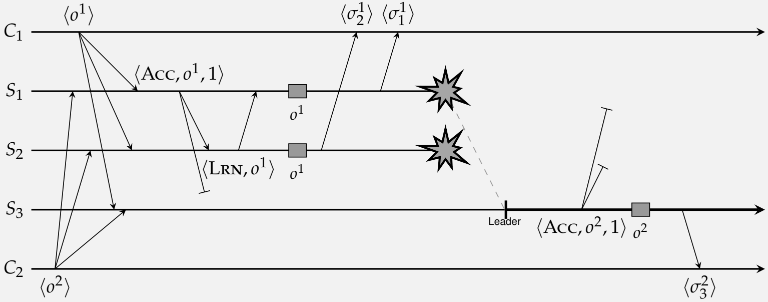
Two servers and one crash: solution


$\langle o^1 \rangle$
$\langle \sigma_2^1 \rangle$
$\langle \sigma_1^1 \rangle$
C_1
$\langle \textsc{Acc}, o^1, 1 \rangle$
S_1
o^1
S_2
Leader
$\langle \textsc{Lrn}, o^1 \rangle$
o^1
$\langle \textsc{Acc}, o^2, 2 \rangle$
o^2
C_2
$\langle o^2 \rangle$
$\langle \sigma_2^{12} \rangle$

Solution:

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

Three servers and two crashes: still a problem?



$\langle o^1 \rangle$
$\langle \sigma_2^1 \rangle$
$\langle \sigma_1^1 \rangle$
C_1
$\langle \textsc{Acc}, o^1, 1 \rangle$
S_1
o^1
S_2
$\langle \textsc{Lrn}, o^1 \rangle$
o^1
S_3
Leader
$\langle \textsc{Acc}, o^2, 1 \rangle$
o^2
C_2
$\langle o^2 \rangle$
$\langle \sigma_3^2 \rangle$

Scenario:

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

Solution:

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

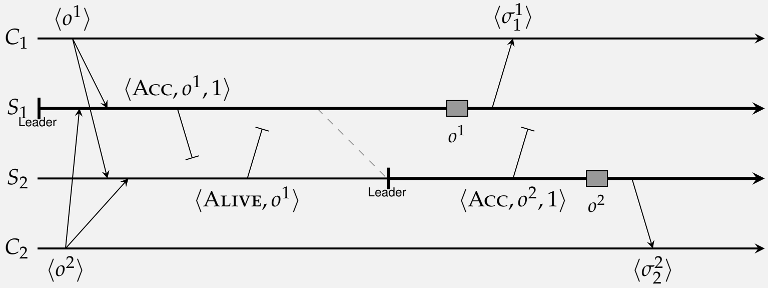
Paxos: fundamental rule

General rule:
In Paxos, a server S cannot execute an operation o until it has received a LEARN(o) from all other nonfaulty servers.

Failure detection

Practice:

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


$\langle o^1 \rangle$
$\langle \sigma_1^1 \rangle$
C_1
$\langle \textsc{Acc}, o^1, 1 \rangle$
S_1
Leader
o^1
S_2
$\langle \textsc{Alive}, o^1 \rangle$
Leader
$\langle \textsc{Acc}, o^2, 1 \rangle$
o^2
C_2
$\langle o^2 \rangle$
$\langle \sigma_2^2 \rangle$

Required number of servers

Observation:

- Paxos needs at least three servers

Adapted fundamental rule:

- In Paxos with three servers, a server S cannot execute an operation o until it has received at least one (other) $\textsc{Learn}(o)$ message, so that it knows that a majority of servers will execute o.

Required number of servers (contd.)

Assumptions before taking the next steps:

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

Observation:

- If either one of the backups (S_2 or S_3) crashes, Paxos will behave correctly: operations at nonfaulty servers are executed in the same order.

Leader crashes after executing o^1

- S_3 is completely ignorant of any activity by S_1
  - S_2 received $\text{ACCEPT}(o, 1)$, detects crash, and becomes leader.
  - S_3 even never received $\text{ACCEPT}(o, 1)$.
  - S_2 sends $\text{ACCEPT}(o^2, 2) \Rightarrow S_3$ sees unexpected timestamp and tells S_2 that it missed o^1.
  - S_2 retransmits $\text{ACCEPT}(o^1, 1)$, allowing S_3 to catch up.
- S_2 missed $\text{ACCEPT}(o^1, 1)$
  - S_2 did detect crash and became new leader
  - S_2 sends $\text{ACCEPT}(o^1, 1) \Rightarrow S_3$ retransmits $\text{LEARN}(o^1)$.
  - S_2 sends $\text{ACCEPT}(o^2, 1) \Rightarrow S_3$ tells S_2 that it apparently missed $\text{ACCEPT}(o^1, 1)$ from S_1, so that S_2 can catch up.

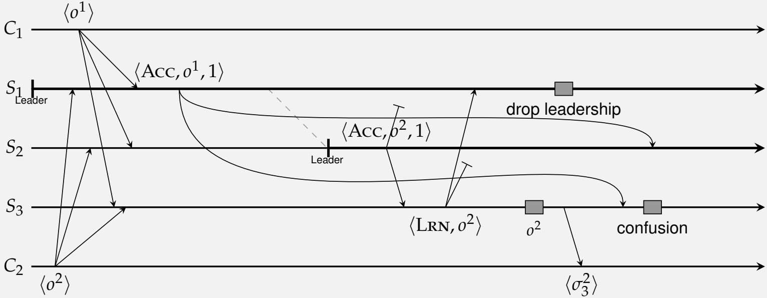
Leader crashes after sending $\textsc{accept}(o^1, 1)$

- S_3 is completely ignorant of any activity by S_1
  - As soon as S_2 announces that o^2 is to be accepted, S_3 will notice that it missed an operation and can ask S_2 to help recover.
- S_2 had missed $\textsc{accept}(o^1, 1)$:
  - As soon as S_2 proposes an operation, it will be using a stale timestamp, allowing S_3 to tell S_2 that it missed operation o^1.

Observation:

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

False crash detection

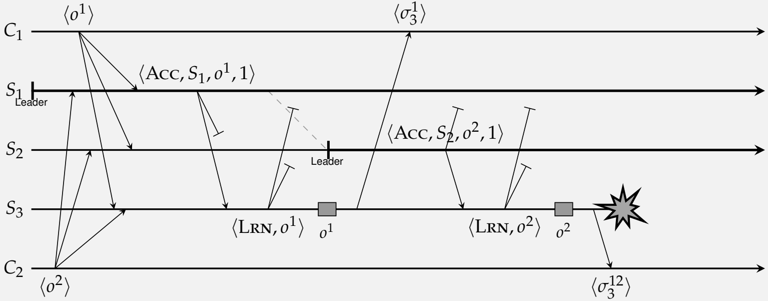


$\langle o^1 \rangle$
C_1
$\langle \text{Acc}, o^1, 1 \rangle$
S_1
Leader
drop leadership
S_2
$\langle \text{Acc}, o^2, 1 \rangle$
Leader
S_3
$\langle \text{Lrn}, o^2 \rangle$
o^2
confusion
C_2
$\langle o^2 \rangle$
$\langle \sigma_3^2 \rangle$

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.

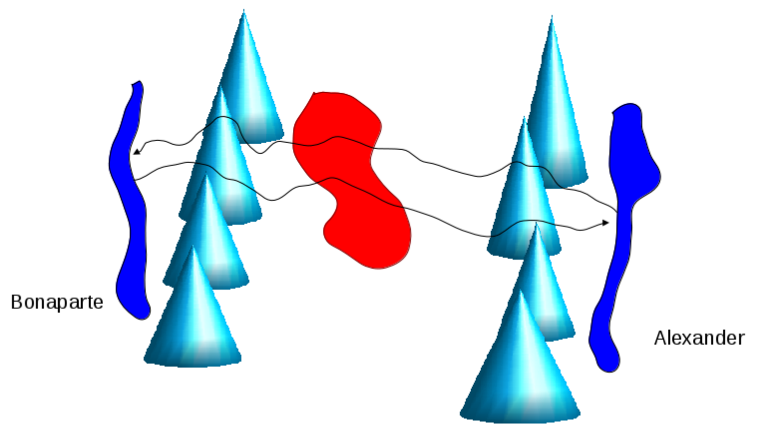
But what about progress (liveness)?


$\langle o^1 \rangle$
$\langle \sigma_3^1 \rangle$
C_1
$\langle \text{Acc}, S_1, o^1, 1 \rangle$
S_1
Leader
S_2
$\langle \text{Acc}, S_2, o^2, 1 \rangle$
Leader
S_3
$\langle \text{Lrn}, o^1 \rangle$
o^1
$\langle \text{Lrn}, o^2 \rangle$
o^2
C_2
$\langle o^2 \rangle$
$\langle \sigma_3^{12} \rangle$

Essence of solution

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

Consensus in Systems with Arbitrary Faults



- Two Army Problem
- Non-faulty generals with unreliable communication.

Byzantine Agreement Problem (1)

Problem:

- Red army in the valley, n blue generals each with their own army surrounding the red army.
- Communication is pairwise, instantaneous and perfect.
- However m of the blue generals are traitors (faulty processes) and are actively trying to prevent the loyal generals from reaching agreement. The generals know the value m.

Goal: The generals need to exchange their troop strengths. At the end of the algorithm, each general has a vector of length n. If ith general is loyal, then the ith element has their correct troop strength otherwise it is undefined.

Conditions for a solution:

- All loyal generals decide upon the same plan of action
- A small number of traitors cannot cause the loyal generals to adopt a bad plan

Byzantine Agreement Problem (2)

Setup:

- Assume that we have n processes, where each process i will provide a value v_i to other processes. The goal is to let each process construct a vector of length n such that if process i is non-faulty, $V[i] = v_i$. Otherwise, $V[i]$ is undefined.
- We assume that there are at most m faulty processes.

Algorithm:

Step 1 Every non-faulty process i sends v_i to every other process using reliable unicasting. Faulty processes may send anything.

- Moreover, since we are using unicasting, the faulty processes may send different values to different processes.

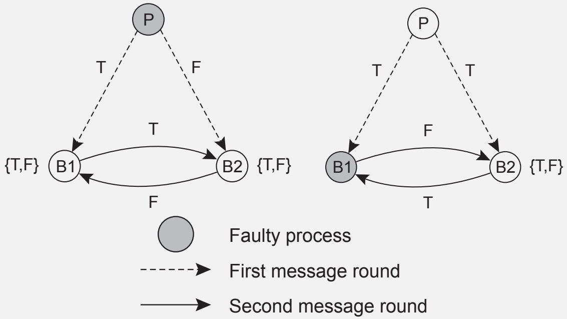
Step 2 The results of the announcements of Step 1 are collected together in the form of a vector of length n.

Step 3 Every process passes its vector from Step 2 to every other process.

Step 4 Each process examines the ith element of each of the newly received vectors. If any value has a majority, that value is put into the result vector. If no value has a majority, the corresponding element of the result vector is set to $\mathrm{UNKNOWN}$.

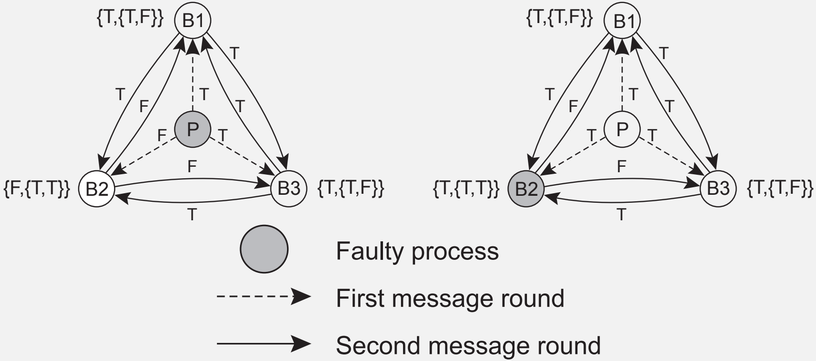
The goal of Byzantine agreement is that consensus is reached on the value for nonfaulty processes only.

Byzantine Agreement Example (1)



- We have 2 loyal generals and one traitor.
- Note that process 1 sees a value of 2 from process 2 (process 2 vector) but sees a value of b for process 2 (process 3 vector). But process 1 has no way of knowing whether process 2 or process 3 is a traitor. So it cannot decide the right value for $V[2]$.
- Similarly, process 2 cannot ascertain the right value for $V[1]$. Hence, they cannot reach agreement.
- For m faulty processes, we need $2m + 1$ non-faulty processes to overcome the traitors. In total, we need a total of $3m + 1$ processes to reach agreement.

Byzantine Agreement Example (2)



The Byzantine generals problem for 3 loyal generals and 1 traitor:

(a) The generals announce their troop strengths (let's say, in units of 1 kilo soldiers).

(b) The vectors that each general assembles based on previous step.

(c) The vectors that each general receives.

(d) If a value has a majority, then we know it correctly, else it is

Limitations on Reaching Consensus (1)

The general goal of distributed consensus algorithms is to have all the non-faulty processes reach consensus on some issue, and to establish that consensus within a finite number of steps. Possible cases:

- Synchronous versus asynchronous systems
- Communication delay is bounded or not
- Message delivery from the same sender is ordered or not
- Message transmission is done through unicasting or multicasting

Limitations on Reaching Consensus (2)

Message ordering

Process behavior	Communication delay	Unordered		Ordered	
		Unicast	Multicast	Unicast	Multicast
Synchronous	Bounded	X	X	X	X
	Unbounded			X	X
Asynchronous	Bounded				X
	Unbounded				X

Message transmission

- Most distributed systems in practice assume that processes behave synchronously, message transmission is unicast, and communication delays are unbounded.

The CAP Theorem

- CAP Theorem (aka *Brewer's Theorem*): states that it is impossible for a distributed computer system to simultaneously provide all three of the following guarantees:
  - **C**onsistency (all nodes see the same data at the same time)
  - **A**vailability (a guarantee that every request receives a response about whether it was successful or failed)
  - **P**artition tolerance (the system continues to operate despite arbitrary message loss or failure of part of the system)

This is a chord, this is another, this is a third. Now form a band!
We know three chords but you can only pick two

Brewer's CAP Theorem (Liberian.com)

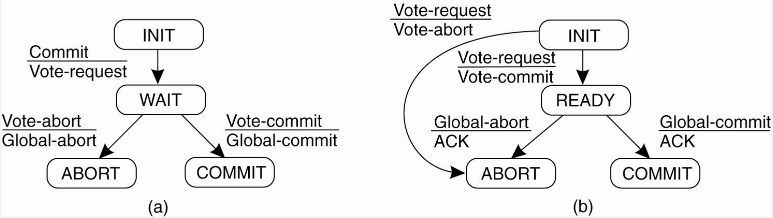
Dealing with CAP

- Drop Partition Tolerance
- Drop Availability
- Drop Consistency
- The BASE Jump: The notion of accepting eventual consistency is supported via an architectural approach known as BASE (**B**asically **A**vailable, **S**oft-state, **E**ventually consistent). BASE, as its name indicates, is the logical opposite of ACID.
- Design around it!

Distributed Commit

- The distributed commit problem involves having an operation being performed by each member of a group or none at all.
- Examples:
  - Reliable multicasting is a specific example with the operation being the delivery of a message.
  - A distributed transaction includes one or more statements that, individually or as a group, update data on two or more distinct nodes of a distributed database.
- Solutions (these all use a coordinator):
  - One-phase commit: Coordinator tells all processes to commit or not. No way for coordinator to know if commit cannot be done locally.
  - Two-phase commit: Coordinator orchestrates the commit using two phases. Does not work if coordinator crashes during the commit process.
  - Three-phase commit: Can work even if the coordinator crashes, but rarely used in practice.

Two Phase Commit (1)



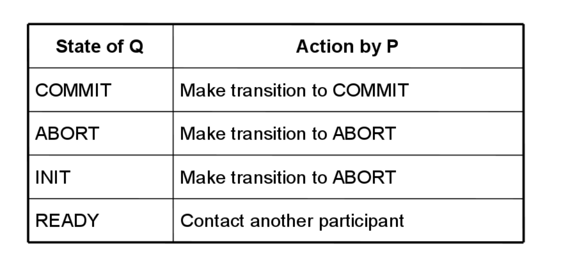
(a) The finite state machine for the coordinator in Two Phase Commit (2PC).

(b) The finite state machine for a participant.

Two Phase Commit (2)

- The protocol can fail if a process crashes for other processes may be waiting indefinitely for a message from the crashed process. We deal with this using timeouts.
- Participant blocked in `INIT`: A participant is waiting for a `VOTE_REQUEST` message from the coordinator. On a timeout, it can locally abort the transaction and thus send a `VOTE_ABORT` message to the coordinator.
- Coordinator blocked in `WAIT`: If it doesn't get all the votes, it votes for an abort and sends a `GLOBAL_ABORT` to all participants.
- Participant blocked in `READY`: Participant cannot simply decide to abort. It needs to know what message was sent by the coordinator.
    - We can simply block until the coordinator recovers.
    - Or contact another participant Q to see if it can decide from Q's state what to do. Four cases to deal with here that are summarized on next slide.

Two Phase Commit (3)



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

- Actions taken by a participant P when residing in state READY and having contacted another participant Q.

Two Phase Commit (4)

- **actions by coordinator**

```
write START_2PC local log;
multicast VOTE_REQUEST  to all participants;
while not all votes have been collected {
    wait for any incoming vote;
    if timeout {
        write GLOBAL_ABORT to local log;
        multicast  GLOBAL_ABORT to all participants;
        exit;
    }
    record vote;
}
if all participants sent VOTE_COMMIT and coordinator votes COMMIT{
    write GLOBAL_COMMIT to local log;
    multicast GLOBAL_COMMIT to all participants;
} else {
    write GLOBAL_ABORT  to local log;
    multicast GLOBAL_ABORT to all participants;
}
```

Two Phase Commit (5)

- **actions by participant**

```
write INIT to local log;
wait for VOTE_REQUEST from coordinator;
if timeout {
    write VOTE_ABORT to local log;
    exit;
}
if participant votes COMMIT {
    write VOTE_COMMIT to local log;
    send VOTE_COMMIT to coordinator;
    wait for DECISION from coordinator;
    if timeout {
        multicast DECISION_REQUEST to other participants;
        wait until DECISION is received; /* remain blocked */
        write DECISION to local log;
    }
    if DECISION == GLOBAL_COMMIT
        write GLOBAL_COMMIT to local log;
    else if DECISION == GLOBAL_ABORT
        write GLOBAL_ABORT to local log;
} else {
    write VOTE_ABORT to local log;
    send  VOTE ABORT to coordinator;
}
```

Two Phase Commit (6)

- **actions for handling decision requests**

```
/* executed by separate thread */
while true {
    wait until any incoming DECISION_REQUEST is received; /* remain b
    read most recently recorded STATE from the local log;
    if STATE == GLOBAL_COMMIT
        send GLOBAL_COMMIT to requesting participant;
    else if STATE == INIT or STATE == GLOBAL_ABORT
        send GLOBAL_ABORT to requesting participant;
    else
        skip;  /* participant remains blocked */
}
```

Recovery

- Backward recovery. Roll back the system from erroneous state to a previously correct state. This requires system to be checkpointing, which has the following issues:
  - Relatively costly to checkpoint. Often combined with message logging for better performance. Messages are logged before sending or before receiving. Combined with checkpoints to makes recovery possible. Checkpoints alone cannot solve the issue of replaying all messages in the right order.
  - Backward recovery requires a loop of recovery so failure transparency cannot be guaranteed. Some states can never be rolled back to...
- Forward recovery. Bring the system to a correct new state from which it can continue execution. E.g. In an (n, k) block erasure code, a set of k source packets is encoded into a set of n encoded packets, such that any set of k encoded packets is enough to reconstruct the original k source packets.

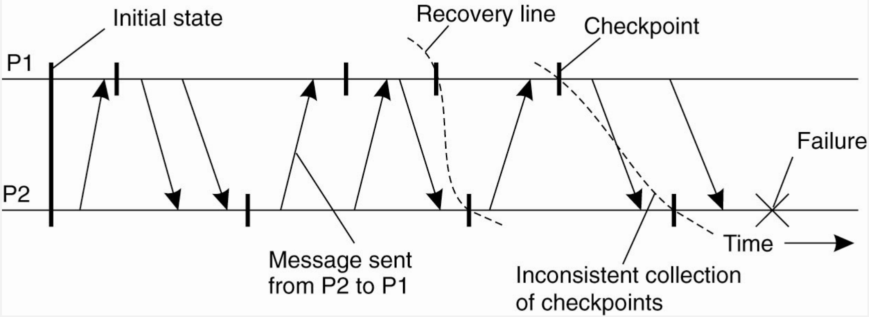
Stable Storage

- We need fault-tolerant disk storage for the checkpoints and message logs. Examples are various RAID (Redundant Array of Independent Disks) schemes (although they are used for both improved fault tolerance as well as improved performance). Some common schemes:
  - RAID-0 block-level striping
  - RAID-1 mirroring
  - RAID-5 block-level striping with distributed parity
  - RAID-6 block-level striping with double distributed parity
  - RAID-10 stripes data across mirrored pairs

Checkpointing

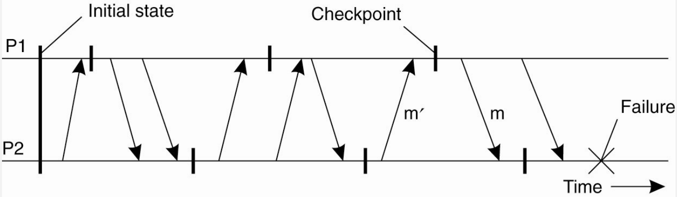
- Backward error recovery schemes require that a distributed system regularly checkpoints a consistent global state to stable storage. This is known as a distributed snapshot.
- In a distributed snapshot, if a process P has recorded the receipt of a message, then there is also a process Q that has recorded the sending of that message.
- To recover after a process or system failure, it is best to recover to the most recent distributed snapshot, also known as the recovery line.
- We will examine the following checkpointing and logging techniques:
  - Independent checkpointing.
  - Coordinated checkpointing.
  - Message logging:
    - Optimistic message logging
    - Pessimistic message logging

Checkpointing and Recovery Line


Initial state
Recovery line
Checkpoint
P1
P2
Failure
Time
Message sent from P2 to P1
Inconsistent collection of checkpoints

Independent Checkpointing

- Each process saves its state from time to time in an uncoordinated fashion.
- To discover a recovery line requires that each process be rolled back to its most recently saved state. If these local states jointly do not form a distributed snapshot, further rolling is necessary. This can lead to a domino effect.
- Find the recovery line in the following timeline:



Coordinated Checkpointing (1)

- All processes synchronize to jointly write their state to local stable storage, which implies that the saved state is automatically consistent.
- Simple Coordinated Checkpointing:
    - Coordinator multicasts a `CHECKPOINT_REQUEST` to all processes.
    - When a process receives the request, it takes a local checkpoint, queues any subsequent messages handed to it by the application it is executing, and acknowledges to the coordinator.
    - When the coordinator has received an acknowledgment from all processes, it multicasts a `CHECKPOINT_DONE` message to allow the blocked processes to continue.

Coordinated Checkpointing (2)

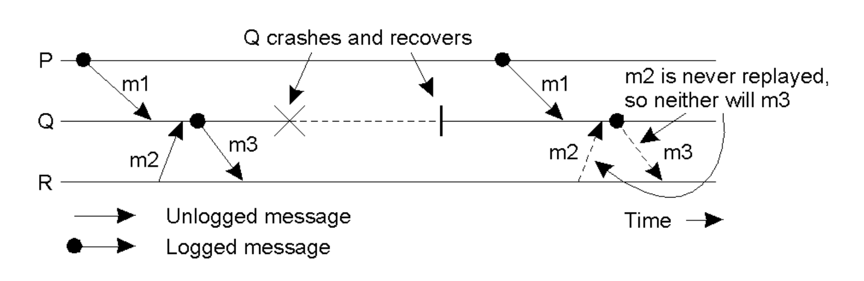
- Incremental Snapshot:
    - The coordinator multicasts a checkpoint request only to those processes it had sent a message to since it last took a checkpoint. The other processes are ignored.
    - When a process P receives such a request, it forwards it to all those processes to which P itself had sent a message since the last checkpoint and so on.
    - A process forwards the request only once.
    - When all processes have been identified, then a second message is multicast to trigger checkpointing and to allow the processes to continue

Message Logging

- Message logging enables us to reduce the number of checkpoints, but still enable recovery.
- If the transmission of messages can be replayed, we can still reach a globally consistent state by starting from a checkpointed state and retransmitting all messages sent since.
- Assumes a piece wise deterministic model, where deterministic intervals occur between sending/receiving messages. The deterministic interval starts with a non-deterministic event such as sending or receiving a message. Then the execution of the process is deterministic until the next non-deterministic event.
- An deterministic interval can be replayed with a known result, provided it is replayed starting with the same non-deterministic event as before. Hence if we record all non-deterministic events, it becomes possible to completely replay the entire execution of a process in a deterministic way.

Orphan Process

- An orphan process is a process that has survived the crash of another process, but whose state is inconsistent with the crashed process after its recovery.
- The figure below shows an incorrect replay of messages after recovery, leading to an orphan process. Note that we assume that m_2 was never logged so it isn't replayed. Which one is the orphan process?



- How to deal with orphans?

References

- "Brewer's CAP Theorem", *julianbrowne.com*, 02-Mar-2010.
- "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services", Nancy Lynch and Seth Gilbert. *ACM SIGACT News*, Volume 33 Issue 2 (2002), pg. 51-59.
- "CAP twelve years later: How the "rules" have changed", Eric Brewer, *IEEE Explore*, Volume 45, Issue 2 (2012), pg. 23-29.
- CAP Theorem Revisited, Robert Greiner, 2014.
- A CAP Solution (proving Brewer Wrong), Guy Pardon's Blog, 2008.
- Your Coffee Shop Doesn't Use Two-Phase Commit, Gregor Hohpe. *IEEE Software*, March/April 2005, pp. 64-66.