Distributed Systems
Principles and Paradigms

Maarten van Steen

VU Amsterdam, Dept. Computer Science
Room R4.20, steen@cs.vu.nl

Chapter 08: Fault Tolerance

Version: December 2, 2009

vrije Universiteit amsterdam

Contents

Introduction

- Basic concepts
- Process resilience
- Reliable client-server communication
- Reliable group communication
- Distributed commit
- Recovery

Dependability

Basics

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

Specifically

A component C depends on C^* if the *correctness* of C's behavior depends on the correctness of C^*'s behavior. Note: components are processes or channels.

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

Terminology

Subtle differences

- Failure: When a component is not living up to its specifications, a failure occurs
- Error: That part of a component's state that can lead to a failure
- Fault: The cause of an error

What to do about faults

- Fault prevention: prevent the occurrence of a fault
- Fault tolerance: build a component such that it can mask the presence of faults
- Fault removal: reduce presence, number, seriousness of faults
- Fault forecasting: estimate present number, future incidence, and consequences of faults

Failure models

Failure semantics

- Crash failures: Component halts, but behaves correctly before halting
- Omission failures: Component fails to respond
- Timing failures: Output is correct, but lies outside a specified real-time interval (performance failures: too slow)
- Response failures: Output is incorrect (but can at least not be accounted to another component)

    Value failure: Wrong value is produced
    State transition failure: Execution of component brings it into a wrong state

- Arbitrary failures: Component produces arbitrary output and be subject to arbitrary timing failures

Crash failures

Problem

Clients cannot distinguish between a crashed component and one that is just a bit slow

Consider a server from which a client is expecting output

- Is the server perhaps exhibiting timing or omission failures?
- Is the channel between client and server faulty?

Assumptions we can make

- Fail-silent: The component exhibits omission or crash failures; clients cannot tell what went wrong
- Fail-stop: The component exhibits crash failures, but its failure can be detected (either through announcement or timeouts)
- Fail-safe: The component exhibits arbitrary, but benign failures (they can't do any harm)

Process resilience

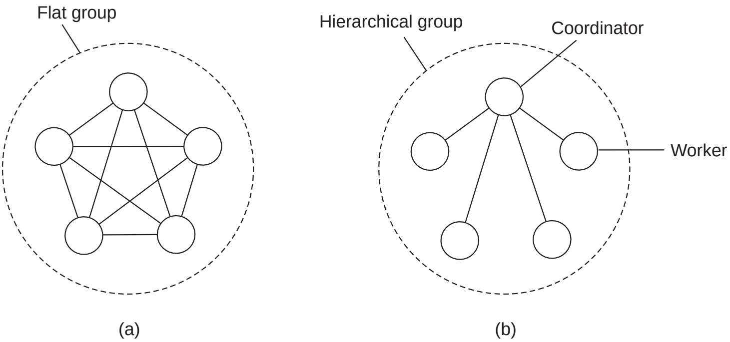
Basic issue

Protect yourself against faulty processes by replicating and distributing computations in a group.

Flat groups: Good for fault tolerance as information exchange immediately occurs with all group members; however, may impose more overhead as control is completely distributed (hard to implement).

Hierarchical groups: All communication through a single coordinator $\Rightarrow$ not really fault tolerant and scalable, but relatively easy to implement.

Process resilience



Groups and failure masking

K-fault tolerant group

When a group can mask any k concurrent member failures (k is called degree of fault tolerance).

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

- Assume crash/performance failure semantics $\Rightarrow$ a total of $k+1$ members are needed to survive k member failures.
- Assume arbitrary failure semantics, and group output defined by voting $\Rightarrow$ a total of $2k+1$ members are needed to survive k member failures.

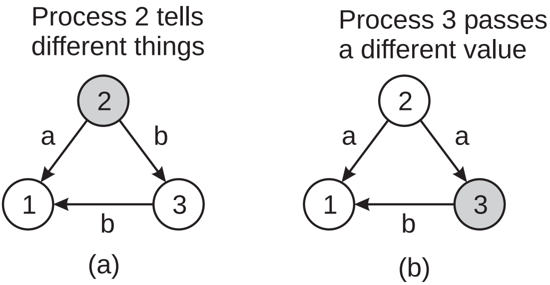
Assumption

All members are identical, and process all input in the same order $\Rightarrow$ only then are we sure that they do exactly the same thing.

Groups and failure masking

Scenario

Group members are not identical, i.e., we have a distributed computation ⇒ Nonfaulty group members should reach agreement on the same value.



Groups and failure masking

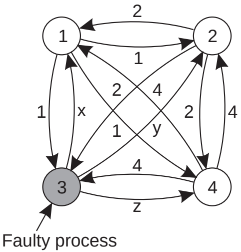
Scenario

Assuming arbitrary failure semantics, we need $3k + 1$ group members to survive the attacks of k faulty members. This is also known as Byzantine failures.

Essence

We are trying to reach a majority vote among the group of loyalists, in the presence of k traitors $\Rightarrow$ need $2k + 1$ loyalists.

Groups and failure masking



(a)

1	Got(1, 2, x, 4)
2	Got(1, 2, y, 4)
3	Got(1, 2, 3, 4)
4	Got(1, 2, z, 4)

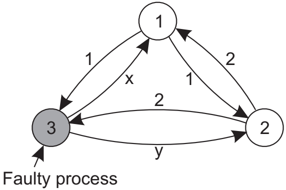
(b)

1 Got	2 Got	4 Got
(1, 2, y, 4)	(1, 2, x, 4)	(1, 2, x, 4)
(a, b, c, d)	(e, f, g, h)	(1, 2, y, 4)
(1, 2, z, 4)	(1, 2, z, 4)	(i, j, k, l)

(c)

- (a) what they send to each other
- (b) what each one got from the other
- (c) what each one got in second step

Groups and failure masking



(a)

1	Got(1, 2, x)
2	Got(1, 2, y)
3	Got(1, 2, 3)

(b)

1 Got	2 Got
(1, 2, y)	(1, 2, x)
(a, b, c)	(d, e, f)

(c)

- (a) what they send to each other
- (b) what each one got from the other
- (c) what each one got in second step

Groups and failure masking

Issue

What are the necessary conditions for reaching agreement?

Message ordering / **Message transmission** / **Process behavior** / **Communication delay**

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

Process:	Synchronous ⇒ operate in lockstep
Delays:	Are delays on communication bounded?
Ordering:	Are messages delivered in the order they were sent?
Transmission:	Are messages sent one-by-one, or multicast?

Failure detection

Essence

We detect failures through timeout mechanisms

- Setting timeouts properly is very difficult and application dependent
- You cannot distinguish process failures from network failures
- We need to consider failure notification throughout the system:
  - Gossiping (i.e., proactively disseminate a failure detection)
  - On failure detection, pretend you failed as well

Reliable communication

So far

Concentrated on process resilience (by means of process groups). What about reliable communication channels?

Error detection

- Framing of packets to allow for bit error detection
- Use of frame numbering to detect packet loss

Error correction

- Add so much redundancy that corrupted packets can be automatically *corrected*
- Request retransmission of lost, or last N packets

Reliable RPC

RPC communication: What can go wrong?

1: Client cannot locate server
2: Client request is lost
3: Server crashes
4: Server response is lost
5: Client crashes

RPC communication: Solutions

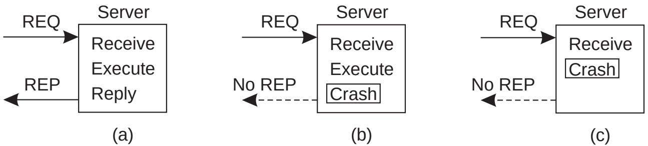
1: Relatively simple – just report back to client
2: Just resend message

Reliable RPC

RPC communication: Solutions

Server crashes

3: Server crashes are harder as you don't what it had already done:



Reliable RPC

Problem

We need to decide on what we expect from the server

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

Reliable RPC

RPC communication: Solutions

Server response is lost

4: Detecting lost replies can be hard, because it can also be that the server had crashed. You don't know whether the server has carried out the operation
Solution: None, except that you can try to make your operations idempotent: repeatable without any harm done if it happened to be carried out before.

Reliable RPC

RPC communication: Solutions

Client crashes

5: Problem: The server is doing work and holding resources for nothing (called doing an orphan computation).

- Orphan is killed (or rolled back) by client when it reboots
- Broadcast new epoch number when recovering ⇒ servers kill orphans
- Require computations to complete in a T time units. Old ones are simply removed.

Question

What's the rolling back for?

Reliable multicasting

Basic model

We have a multicast channel c with two (possibly overlapping) groups:

- The sender group $\text{SND}(c)$ of processes that *submit* messages to channel c
- The receiver group $\text{RCV}(c)$ of processes that can receive messages from channel c

Simple reliability: If process $P \in \text{RCV}(c)$ at the time message m was submitted to c, and P does not leave $\text{RCV}(c)$, m should be delivered to P

Atomic multicast: How can we ensure that a message m submitted to channel c is delivered to process $P \in \text{RCV}(c)$ only if m is delivered to *all* members of $\text{RCV}(c)$

Reliable multicasting

Observation

If we can stick to a local-area network, reliable multicasting is “easy”

Principle

Let the sender log messages submitted to channel c:

- If P sends message m, m is stored in a history buffer
- Each receiver acknowledges the receipt of m, or requests retransmission at P when noticing message lost
- Sender P removes m from history buffer when everyone has acknowledged receipt

Question

Why doesn’t this scale?

Scalable reliable multicasting: Feedback suppression

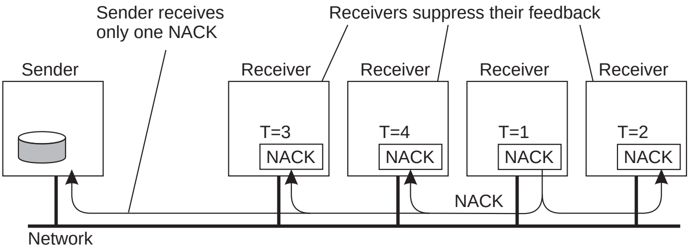
Basic idea

Let a process P suppress its own feedback when it notices another process Q is already asking for a retransmission

Assumptions

- All receivers listen to a common feedback channel to which feedback messages are submitted
- Process P schedules its own feedback message *randomly*, and suppresses it when observing another feedback message

Scalable reliable multicasting: Feedback suppression



Question

Why is the random schedule so important?

Scalable reliable multicasting: Hierarchical solutions

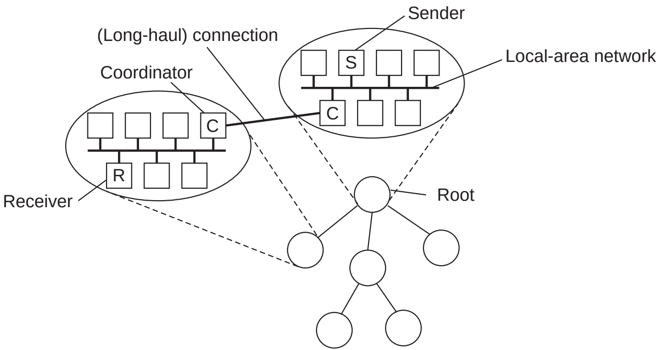
Basic solution

Construct a hierarchical feedback channel in which all submitted messages are sent only to the root. Intermediate nodes aggregate feedback messages before passing them on.

Observation

Intermediate nodes can easily be used for retransmission purposes

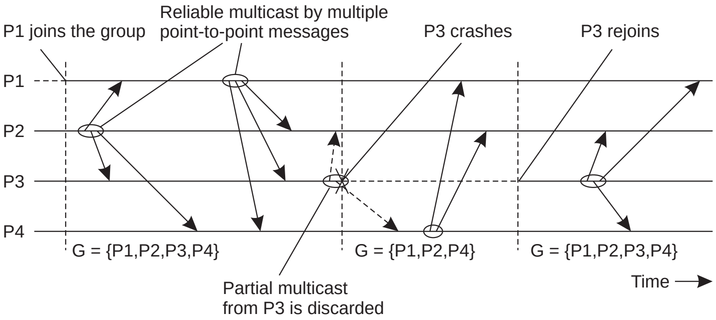
Scalable reliable multicasting: Hierarchical solutions



Question

What's the main problem with this solution?

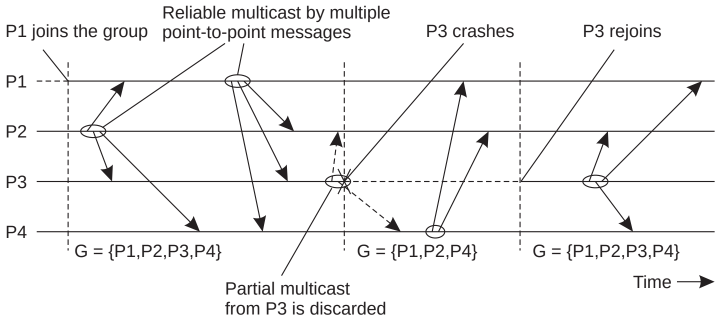
Atomic multicast



Idea

Formulate reliable multicasting in the presence of process failures in terms of process groups and changes to group membership.

Atomic multicast



Guarantee

A message is delivered only to the nonfaulty members of the current group. All members should agree on the current group membership ⇒ Virtually synchronous multicast.

Virtual synchrony

Essence

We consider views $V \subseteq \text{RCV}(c) \cup \text{SND}(c)$

Principle

Processes are added or deleted from a view V through view changes to V^*; a view change is to be executed *locally* by each $P \in V \cap V^*$

(1) For each consistent state, there is a unique view on which all its members agree. Note: implies that all nonfaulty processes see all view changes in the same order

Virtual synchrony

Principle (cnt'd)

(2) If message m is sent to V before a view change vc to V^*, then either all $P \in V$ that excute vc receive m, or no processes $P \in V$ that execute vc receive m. Note: all nonfaulty members in the same view get to see the same set of multicast messages.

(3) A message sent to view V can be delivered only to processes in V, and is discarded by successive views

Definition

A reliable multicast algorithm satisfying (1)–(3) is virtually synchronous

Virtual synchrony

How it works

- A sender to a view V need not be member of V
- If a sender $S \in V$ crashes, its multicast message m is *flushed* before S is removed from V: m will never be delivered after the point that $S \notin V$

  Note: Messages from S may still be delivered to all, or none (nonfaulty) processes in V before they all agree on a new view to which S does not belong
- If a receiver P fails, a message m may be lost but can be recovered as we know exactly what has been received in V. Alternatively, we may decide to deliver m to members in $V - \{P\}$

Virtual synchrony

How it works

- A sender to a view V need not be member of V
- If a sender $S \in V$ crashes, its multicast message m is *flushed* before S is removed from V: m will never be delivered after the point that $S \notin V$
  Note: Messages from S may still be delivered to all, or none (nonfaulty) processes in V before they all agree on a new view to which S does not belong
- If a receiver P fails, a message m may be lost but can be recovered as we know exactly what has been received in V. Alternatively, we may decide to deliver m to members in $V - \{P\}$

Virtual synchrony

How it works

- A sender to a view V need not be member of V
- If a sender $S \in V$ crashes, its multicast message m is *flushed* before S is removed from V: m will never be delivered after the point that $S \notin V$
  Note: Messages from S may still be delivered to all, or none (nonfaulty) processes in V before they all agree on a new view to which S does not belong
- If a receiver P fails, a message m may be lost but can be recovered as we know exactly what has been received in V. Alternatively, we may decide to deliver m to members in $V - \{P\}$

Virtual synchrony

How it works

- A sender to a view V need not be member of V
- If a sender $S \in V$ crashes, its multicast message m is *flushed* before S is removed from V: m will never be delivered after the point that $S \notin V$

  Note: Messages from S may still be delivered to all, or none (nonfaulty) processes in V before they all agree on a new view to which S does not belong
- If a receiver P fails, a message m may be lost but can be recovered as we know exactly what has been received in V. Alternatively, we may decide to deliver m to members in $V - \{P\}$

Virtual synchrony

Observation

Virtually synchronous behavior can be seen independent from the ordering of message delivery. The only issue is that messages are delivered to an *agreed upon* group of receivers.

Virtual synchrony implementation

Some gory details...

- The current view is known at each P by means of a delivery list *dest[P]*
- If $P \in$ *dest[Q]* then $Q \in$ *dest[P]*
- Messages received by P are queued in *queue[P]*
- If P fails, the group view must change, but not before all messages from P have been flushed
- Each P attaches a (stepwise increasing) timestamp with each message it sends
- Assume FIFO-ordered delivery; the highest numbered message from Q that has been received by P is recorded in *rcvd[P][Q]*
- The vector *rcvd[P][]* is sent (as a control message) to all members in *dest[P]*
- Each P records *rcvd[Q][]* in *remote[P][Q]*

Virtual synchrony implementation

Some gory details...

- The current view is known at each P by means of a delivery list *dest[P]*
- If $P \in$ *dest[Q]* then $Q \in$ *dest[P]*
- Messages received by P are queued in *queue[P]*
- If P fails, the group view must change, but not before all messages from P have been flushed
- Each P attaches a (stepwise increasing) timestamp with each message it sends
- Assume FIFO-ordered delivery; the highest numbered message from Q that has been received by P is recorded in *rcvd[P][Q]*
- The vector *rcvd[P][]* is sent (as a control message) to all members in *dest[P]*
- Each P records *rcvd[Q][]* in *remote[P][Q]*

Virtual synchrony implementation

Some gory details...

- The current view is known at each *P* by means of a delivery list *dest[P]*
- If $P \in$ *dest[Q]* then $Q \in$ *dest[P]*
- Messages received by *P* are queued in *queue[P]*
- If *P* fails, the group view must change, but not before all messages from *P* have been flushed
- Each *P* attaches a (stepwise increasing) timestamp with each message it sends
- Assume FIFO-ordered delivery; the highest numbered message from *Q* that has been received by *P* is recorded in *rcvd[P][Q]*
- The vector *rcvd[P][]* is sent (as a control message) to all members in *dest[P]*
- Each *P* records *rcvd[Q][]* in *remote[P][Q]*

Virtual synchrony implementation

Some gory details...

- The current view is known at each P by means of a delivery list *dest[P]*
- If $P \in$ *dest[Q]* then $Q \in$ *dest[P]*
- Messages received by P are queued in *queue[P]*
- If P fails, the group view must change, but not before all messages from P have been flushed
- Each P attaches a (stepwise increasing) timestamp with each message it sends
- Assume FIFO-ordered delivery; the highest numbered message from Q that has been received by P is recorded in *rcvd[P][Q]*
- The vector *rcvd[P][]* is sent (as a control message) to all members in *dest[P]*
- Each P records *rcvd[Q][]* in *remote[P][Q]*

Virtual synchrony implementation

Some gory details...

- The current view is known at each P by means of a delivery list *dest[P]*
- If $P \in$ *dest[Q]* then $Q \in$ *dest[P]*
- Messages received by P are queued in *queue[P]*
- If P fails, the group view must change, but not before all messages from P have been flushed
- Each P attaches a (stepwise increasing) timestamp with each message it sends
- Assume FIFO-ordered delivery; the highest numbered message from Q that has been received by P is recorded in *rcvd[P][Q]*
- The vector *rcvd[P][]* is sent (as a control message) to all members in *dest[P]*
- Each P records *rcvd[Q][]* in *remote[P][Q]*

Virtual synchrony implementation

Some gory details...

- The current view is known at each P by means of a delivery list *dest[P]*
- If $P \in$ *dest[Q]* then $Q \in$ *dest[P]*
- Messages received by P are queued in *queue[P]*
- If P fails, the group view must change, but not before all messages from P have been flushed
- Each P attaches a (stepwise increasing) timestamp with each message it sends
- Assume FIFO-ordered delivery; the highest numbered message from Q that has been received by P is recorded in *rcvd[P][Q]*
- The vector *rcvd[P][]* is sent (as a control message) to all members in *dest[P]*
- Each P records *rcvd[Q][]* in *remote[P][Q]*

Virtual synchrony implementation

Some gory details...

- The current view is known at each P by means of a delivery list *dest[P]*
- If $P \in$ *dest[Q]* then $Q \in$ *dest[P]*
- Messages received by P are queued in *queue[P]*
- If P fails, the group view must change, but not before all messages from P have been flushed
- Each P attaches a (stepwise increasing) timestamp with each message it sends
- Assume FIFO-ordered delivery; the highest numbered message from Q that has been received by P is recorded in *rcvd[P][Q]*
- The vector *rcvd[P][]* is sent (as a control message) to all members in *dest[P]*
- Each P records *rcvd[Q][]* in *remote[P][Q]*

Virtual synchrony implementation

Some gory details...

- The current view is known at each P by means of a delivery list *dest[P]*
- If $P \in$ *dest[Q]* then $Q \in$ *dest[P]*
- Messages received by P are queued in *queue[P]*
- If P fails, the group view must change, but not before all messages from P have been flushed
- Each P attaches a (stepwise increasing) timestamp with each message it sends
- Assume FIFO-ordered delivery; the highest numbered message from Q that has been received by P is recorded in *rcvd[P][Q]*
- The vector *rcvd[P][]* is sent (as a control message) to all members in *dest[P]*
- Each P records *rcvd[Q][]* in *remote[P][Q]*

Virtual synchrony implementation

Some gory details...

- The current view is known at each P by means of a delivery list *dest[P]*
- If $P \in$ *dest[Q]* then $Q \in$ *dest[P]*
- Messages received by P are queued in *queue[P]*
- If P fails, the group view must change, but not before all messages from P have been flushed
- Each P attaches a (stepwise increasing) timestamp with each message it sends
- Assume FIFO-ordered delivery; the highest numbered message from Q that has been received by P is recorded in *rcvd[P][Q]*
- The vector *rcvd[P][]* is sent (as a control message) to all members in *dest[P]*
- Each P records *rcvd[Q][]* in *remote[P][Q]*

Virtual synchrony implementation

Observation

remote[P][Q] shows what *P* knows about message arrival at *Q*

1	2	3	1	5
2	2	2	2	4
3	3	1	4	5
4	4	2	2	4
min	2	1	1	4

Virtual synchrony implementation

Prnciple

- A message is stable if it has been received by all $Q \in dest[P]$ (shown as the min vector)
- Stable messages can be delivered to the next layer (which may deal with ordering). Note: Causal message delivery comes for free
- As soon as all messages from the faulty process have been flushed, that process can be removed from the (local) views

Virtual synchrony implementation

Remains

What if a sender P failed and not all its messages made it to the nonfaulty members of the current view?

Solution

Select a coordinator which has all (unstable) messages from P, and forward those to the other group members.

Note

Member failure is assumed to be detected and subsequently multicast to the current view as a view change. That view change will not be carried out before all messages in the current view have been delivered.

Virtual synchrony implementation

Remains

What if a sender P failed and not all its messages made it to the nonfaulty members of the current view?

Solution

Select a coordinator which has all (unstable) messages from P, and forward those to the other group members.

Note

Member failure is assumed to be detected and subsequently multicast to the current view as a view change. That view change will not be carried out before all messages in the current view have been delivered.

Virtual synchrony implementation

Remains

What if a sender *P* failed and not all its messages made it to the nonfaulty members of the current view?

Solution

Select a coordinator which has all (unstable) messages from *P*, and forward those to the other group members.

Note

Member failure is assumed to be detected and subsequently multicast to the current view as a view change. That view change will not be carried out before all messages in the current view have been delivered.

Distributed commit

- Two-phase commit
- Three-phase commit

Essential issue

Given a computation distributed across a process group, how can we ensure that either all processes commit to the final result, or none of them do (atomicity)?

Distributed commit

- Two-phase commit
- Three-phase commit

Essential issue

Given a computation distributed across a process group, how can we ensure that either all processes commit to the final result, or none of them do (atomicity)?

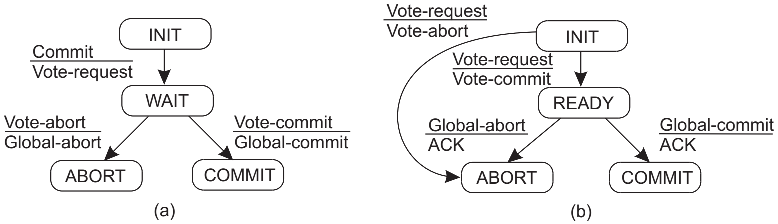
Two-phase commit

Model

The client who inititated the computation acts as coordinator; processes required to commit are the participants

- Phase 1a: Coordinator sends *vote-request* to participants (also called a pre-write)
- Phase 1b: When participant receives *vote-request* it returns either *vote-commit* or *vote-abort* to coordinator. If it sends *vote-abort*, it aborts its local computation
- Phase 2a: Coordinator collects all votes; if all are *vote-commit*, it sends *global-commit* to all participants, otherwise it sends *global-abort*
- Phase 2b: Each participant waits for *global-commit* or *global-abort* and handles accordingly.

Two-phase commit



Coordinator Participant

2PC – Failing participant

Scenario

Participant crashes in state S, and recovers to S

- Initial state: No problem: participant was unaware of protocol
- Ready state: Participant is waiting to either commit or abort. After recovery, participant needs to know which state transition it should make $\Rightarrow$ log the coordinator's decision
- Abort state: Merely make entry into abort state *idempotent*, e.g., removing the workspace of results
- Commit state: Also make entry into commit state idempotent, e.g., copying workspace to storage.

Observation

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

2PC – Failing participant

Scenario

Participant crashes in state S, and recovers to S

- Initial state: No problem: participant was unaware of protocol
- Ready state: Participant is waiting to either commit or abort. After recovery, participant needs to know which state transition it should make $\Rightarrow$ log the coordinator's decision
- Abort state: Merely make entry into abort state *idempotent*, e.g., removing the workspace of results
- Commit state: Also make entry into commit state idempotent, e.g., copying workspace to storage.

Observation

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

2PC – Failing participant

Scenario

Participant crashes in state S, and recovers to S

- Initial state: No problem: participant was unaware of protocol
- Ready state: Participant is waiting to either commit or abort. After recovery, participant needs to know which state transition it should make $\Rightarrow$ log the coordinator's decision
- Abort state: Merely make entry into abort state *idempotent*, e.g., removing the workspace of results
- Commit state: Also make entry into commit state idempotent, e.g., copying workspace to storage.

Observation

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

2PC – Failing participant

Scenario

Participant crashes in state S, and recovers to S

- Initial state: No problem: participant was unaware of protocol
- Ready state: Participant is waiting to either commit or abort. After recovery, participant needs to know which state transition it should make $\Rightarrow$ log the coordinator's decision
- Abort state: Merely make entry into abort state *idempotent*, e.g., removing the workspace of results
- Commit state: Also make entry into commit state idempotent, e.g., copying workspace to storage.

Observation

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

2PC – Failing participant

Scenario

Participant crashes in state S, and recovers to S

- Initial state: No problem: participant was unaware of protocol
- Ready state: Participant is waiting to either commit or abort. After recovery, participant needs to know which state transition it should make $\Rightarrow$ log the coordinator's decision
- Abort state: Merely make entry into abort state *idempotent*, e.g., removing the workspace of results
- Commit state: Also make entry into commit state idempotent, e.g., copying workspace to storage.

Observation

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

2PC – Failing participant

Alternative

When a recovery is needed to READY state, check state of other participants $\Rightarrow$ 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.

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

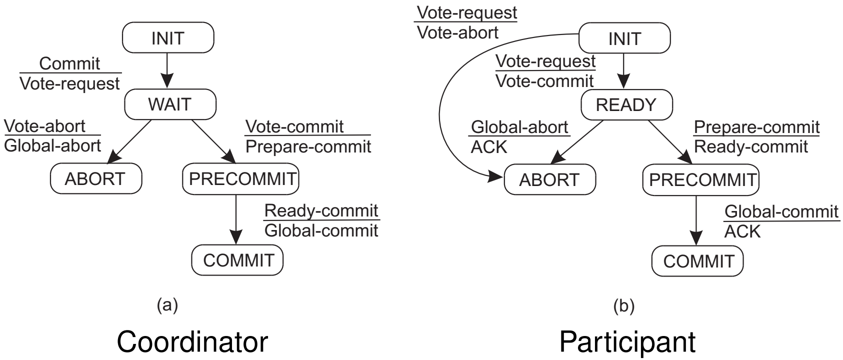
Essence of the problem is that a recovering participant cannot make a local decision: it is dependent on other (possibly failed) processes

Three-phase commit

Model (Again: the client acts as coordinator)

- Phase 1a: Coordinator sends *vote-request* to participants
- 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 *prepare-commit* to all participants, otherwise it sends *global-abort*, and halts
- Phase 2b: Each participant waits for *prepare-commit*, or waits for *global-abort* after which it halts
- Phase 3a: (Prepare to commit) Coordinator waits until all participants have sent *ready-commit*, and then sends *global-commit* to all
- Phase 3b: (Prepare to commit) Participant waits for *global-commit*

Three-phase commit



3PC – Failing participant

Basic issue

Can *P* find out what it should it do after crashing in the ready or pre-commit state, even if other participants or the coordinator failed?

Reasoning

Essence: Coordinator and participants on their way to commit, never differ by more than one state transition

Consequence: If a participant timeouts in ready state, it can find out at the coordinator or other participants whether it should abort, or enter pre-commit state

Observation: If a participant already made it to the pre-commit state, it can always safely commit (but is not allowed to do so for the sake of failing other processes)

Observation: We may need to elect another coordinator to send off the final *COMMIT*

Recovery

- Introduction
- Checkpointing
- Message Logging

Recovery: Background

Essence

When a failure occurs, we need to bring the system into an error-free state:

- Forward error recovery: Find a new state from which the system can continue operation
- Backward error recovery: Bring the system back into a *previous* error-free state

Practice

Use backward error recovery, requiring that we establish recovery points

Observation

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

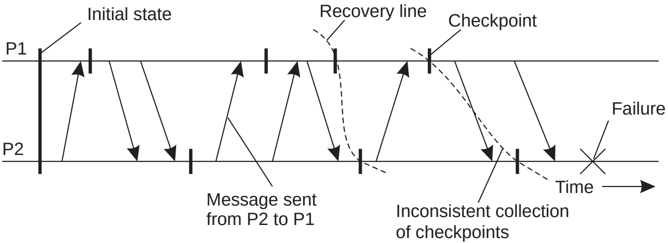
Consistent recovery state

Requirement

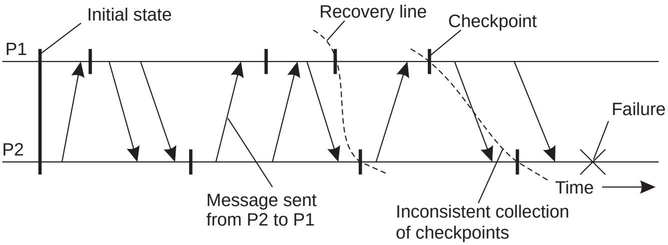
Every message that has been received is also shown to have been sent in the state of the sender.

Recovery line

Assuming processes regularly checkpoint their state, the most recent consistent global checkpoint.



Consistent recovery state



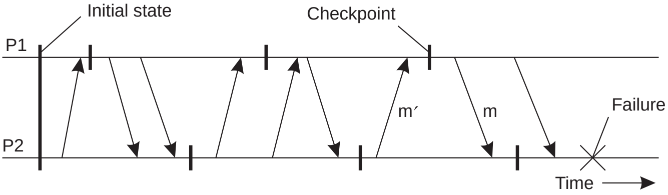
Observation

If and only if the system provides *reliable* communication, should sent messages also be received in a consistent state.

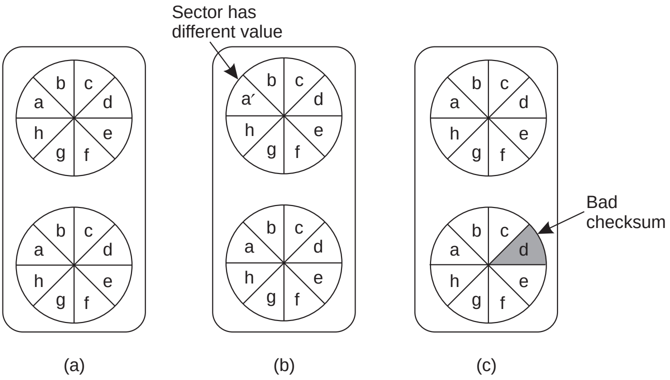
Cascaded rollback

Observation

If checkpointing is done at the “wrong” instants, the recovery line may lie at system startup time ⇒ cascaded rollback



Checkpointing: Stable storage



After a crash

- If both disks are identical: you're in good shape.
- If one is bad, but the other is okay (checksums): choose the good one.
- If both seem okay, but are different: choose the main disk.
- If both aren't good: you're not in a good shape.

Independent checkpointing

Essence

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

- Let $CP[i](m)$ denote m^{th} checkpoint of process P_i and $INT[i](m)$ the interval between $CP[i](m-1)$ and $CP[i](m)$
- When process P_i sends a message in interval $INT[i](m)$, it piggybacks (i,m)
- When process P_j receives a message in interval $INT[j](n)$, it records the dependency $INT[i](m) \rightarrow INT[j](n)$
- The dependency $INT[i](m) \rightarrow INT[j](n)$ is saved to stable storage when taking checkpoint $CP[j](n)$

Independent checkpointing

Essence

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

- Let $CP[i](m)$ denote m^{th} checkpoint of process P_i and $INT[i](m)$ the interval between $CP[i](m-1)$ and $CP[i](m)$
- When process P_i sends a message in interval $INT[i](m)$, it piggybacks (i, m)
- When process P_j receives a message in interval $INT[j](n)$, it records the dependency $INT[i](m) \rightarrow INT[j](n)$
- The dependency $INT[i](m) \rightarrow INT[j](n)$ is saved to stable storage when taking checkpoint $CP[j](n)$

Independent checkpointing

Essence

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

- Let $CP[i](m)$ denote m^{th} checkpoint of process P_i and $INT[i](m)$ the interval between $CP[i](m-1)$ and $CP[i](m)$
- When process P_i sends a message in interval $INT[i](m)$, it piggybacks (i,m)
- When process P_j receives a message in interval $INT[j](n)$, it records the dependency $INT[i](m) \rightarrow INT[j](n)$
- The dependency $INT[i](m) \rightarrow INT[j](n)$ is saved to stable storage when taking checkpoint $CP[j](n)$

Independent checkpointing

Essence

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

- Let $CP[i](m)$ denote m^{th} checkpoint of process P_i and $INT[i](m)$ the interval between $CP[i](m-1)$ and $CP[i](m)$
- When process P_i sends a message in interval $INT[i](m)$, it piggybacks (i,m)
- When process P_j receives a message in interval $INT[j](n)$, it records the dependency $INT[i](m) \rightarrow INT[j](n)$
- The dependency $INT[i](m) \rightarrow INT[j](n)$ is saved to stable storage when taking checkpoint $CP[j](n)$

Independent checkpointing

Essence

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

- Let $CP[i](m)$ denote m^{th} checkpoint of process P_i and $INT[i](m)$ the interval between $CP[i](m-1)$ and $CP[i](m)$
- When process P_i sends a message in interval $INT[i](m)$, it piggybacks (i,m)
- When process P_j receives a message in interval $INT[j](n)$, it records the dependency $INT[i](m) \rightarrow INT[j](n)$
- The dependency $INT[i](m) \rightarrow INT[j](n)$ is saved to stable storage when taking checkpoint $CP[j](n)$

Independent checkpointing

Observation

If process P_i rolls back to $CP[i](m-1)$, P_j must roll back to $CP[j](n-1)$.

Question

How can P_j find out where to roll back to?

Coordinated checkpointing

Essence

Each process takes a checkpoint after a globally coordinated action.

Question

What advantages are there to coordinated checkpointing?

Coordinated checkpointing

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

Coordinated checkpointing

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

Coordinated checkpointing

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

Coordinated checkpointing

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

Coordinated checkpointing

Simple solution

Use a two-phase blocking protocol:

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

Observation

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

Message logging

Alternative

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

Assumption

We assume a piecewise deterministic execution model:

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

Message logging

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.

Question

Why is logging only *messages* not enough?

Question

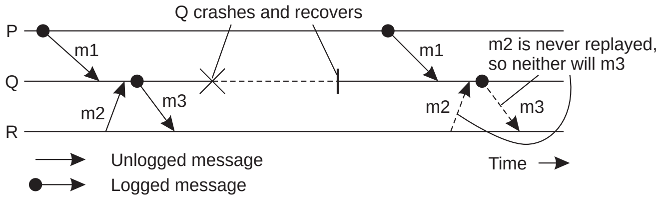
Is logging only nondeterministic events enough?

Message logging and consistency

When should we actually log messages?

Issue: Avoid orphans:

- Process Q has just received and subsequently 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



Message-logging schemes

Notations

HDR[m]: The header of message *m* containing its source, destination, sequence number, and delivery number

The header contains all information for resending a message and delivering it in the correct order (assume data is reproduced by the application)

A message *m* is stable if *HDR[m]* cannot be lost (e.g., because it has been written to stable storage)

DEP[m]: The set of processes to which message *m* has been delivered, as well as any message that causally depends on delivery of *m*

COPY[m]: The set of processes that have a copy of *HDR[m]* in their volatile memory

Message-logging schemes

Characterization

If C is a collection of crashed processes, then $Q \notin C$ is an orphan if there is a message m such that $Q \in DEP[m]$ and $COPY[m] \subseteq C$

Message-logging schemes

Note

We want $\forall m \forall C :: COPY[m] \subseteq C \Rightarrow DEP[m] \subseteq C$. This is the same as saying that $\forall m :: DEP[m] \subseteq COPY[m]$.

Goal

No orphans means that for each message m,

$$DEP[m] \subseteq COPY[m]$$

Message-logging schemes

Note

We want $\forall m \forall C :: COPY[m] \subseteq C \Rightarrow DEP[m] \subseteq C$. This is the same as saying that $\forall m :: DEP[m] \subseteq COPY[m]$.

Goal

No orphans means that for each message m,

$$DEP[m] \subseteq COPY[m]$$

Message-logging schemes

Pessimistic protocol

For each *nonstable* message m, there is at most one process dependent on m, that is $|DEP[m]| \leq 1$.

Consequence

An unstable message in a pessimistic protocol *must* be made stable before sending a next message.

Message-logging schemes

Optimistic protocol

For each unstable message m, we ensure that if $COPY[m] \subseteq C$, then eventually also $DEP[m] \subseteq C$, where C denotes a set of processes that have been marked as faulty

Consequence

To guarantee that $DEP[m] \subseteq C$, we generally rollback each orphan process Q until $Q \notin DEP[m]$