Distributed Systems
Principles and Paradigms

Maarten van Steen

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

Chapter 08: Fault Tolerance

Version: December 11, 2012

vrije Universiteit amsterdam

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

Fault Tolerance 8.3 Reliable Communication

2 / 35

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

Fault Tolerance 8.3 Reliable Communication

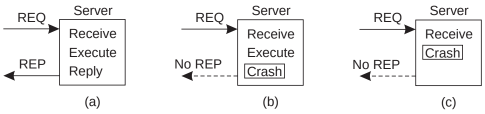
3 / 35

Reliable RPC

RPC communication: Solutions

Server crashes

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



Fault Tolerance 8.3 Reliable Communication

4 / 35

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.

Fault Tolerance 8.3 Reliable Communication

5 / 35

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.

Fault Tolerance 8.3 Reliable Communication

6 / 35

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?

Fault Tolerance 8.3 Reliable Communication

Reliable multicasting

Basic model

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

- The sender group SND(c) of processes that *submit* messages to channel c
- The receiver group 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 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 RCV(c)

Fault Tolerance 8.4 Reliable Group Communication

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:

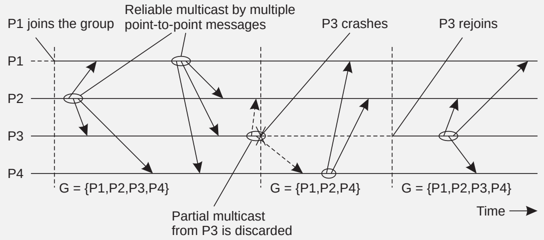
- If P sends message m, m is stored in a history buffer
- Each receiver acknowledges the receipt of m, or requests retransmission at P when noticing message lost
- Sender P removes m from history buffer when everyone has acknowledged receipt

Question

Why doesn't this scale?

Fault Tolerance 8.4 Reliable Group Communication

Atomic multicast



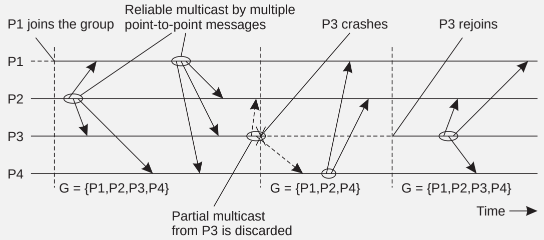
Idea

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

Fault Tolerance 8.4 Reliable Group Communication

10 / 35

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.

Fault Tolerance 8.4 Reliable Group Communication

11 / 35

Atomic multicast vs. Paxos

Question

How can Paxos be used to realize atomic multicast?

Fault Tolerance 8.4 Reliable Group Communication

12 / 35

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)?

Fault Tolerance 8.5 Distributed Commit

13 / 35

Two-phase commit

Model

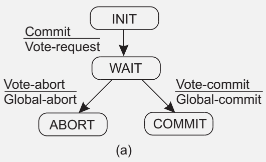
The client who initiated the computation acts as coordinator; processes required to commit are the participants

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

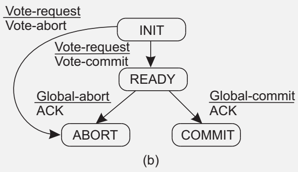
Fault Tolerance 8.5 Distributed Commit

14 / 35

Two-phase commit



Coordinator



Participant

Fault Tolerance 8.5 Distributed Commit

15 / 35

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.

Fault Tolerance 8.5 Distributed Commit

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.

Fault Tolerance 8.5 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

Fault Tolerance 8.5 Distributed Commit

Recovery

- Introduction
- Checkpointing
- Message Logging

Fault Tolerance 8.6 Recovery

19 / 35

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

Fault Tolerance 8.6 Recovery

20 / 35

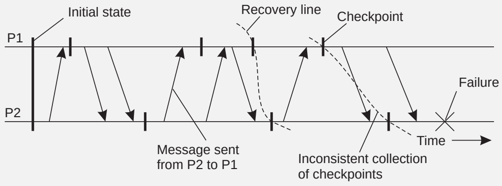
Consistent recovery state

Requirement

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

Recovery line

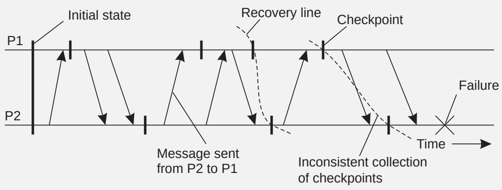
Assuming processes regularly checkpoint their state, the most recent consistent global checkpoint.



Fault Tolerance 8.6 Recovery

21 / 35

Consistent recovery state



Observation

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

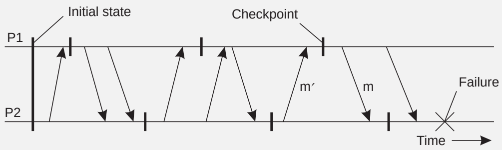
Fault Tolerance 8.6 Recovery

22 / 35

Cascaded rollback

Observation

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



Fault Tolerance 8.6 Recovery

23 / 35

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 $CP[j](n)$

Fault Tolerance 8.6 Recovery

24 / 35

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?

Fault Tolerance 8.6 Recovery

25 / 35

Coordinated checkpointing

Essence

Each process takes a checkpoint after a globally coordinated action.

Question

What advantages are there to coordinated checkpointing?

Fault Tolerance 8.6 Recovery

26 / 35

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

Fault Tolerance 8.6 Recovery

27 / 35

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

Fault Tolerance 8.6 Recovery

28 / 35

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?

Fault Tolerance 8.6 Recovery

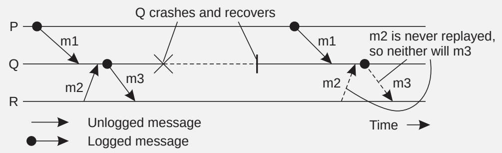
29 / 35

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 (and becomes an orphan when Q recovers).



Fault Tolerance 8.6 Recovery

30 / 35

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 safely written to storage) .

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

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

Fault Tolerance 8.6 Recovery

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 $COPY[m] \subseteq C$

Fault Tolerance 8.6 Recovery

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,

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

Fault Tolerance 8.6 Recovery

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.

Observation

The single recipient of m can safely crash without this leading to orphans: $\forall m :: DEP[m] \subseteq COPY[m]$.

Fault Tolerance 8.6 Recovery

34 / 35

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 $Q \notin DEP[m]$.

Fault Tolerance 8.6 Recovery

35 / 35