Distributed Systems
Principles and Paradigms

Maarten van Steen

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

Chapter 07: Consistency & Replication

Version: November 26, 2012

Distribution protocols

- Replica server placement
- Content replication and placement
- Content distribution

single data item

Replica placement

Essence

Figure out what the best K places are out of N possible locations.

- Select best location out of $N - K$ for which the average distance to clients is minimal. Then choose the next best server. (Note: The first chosen location minimizes the average distance to all clients.) Computationally expensive. $O(N^2)$
- Select the K-th largest autonomous system and place a server at the best-connected host. Computationally expensive. $O(N^2)$
- Position nodes in a d-dimensional geometric space, where distance reflects latency. Identify the K regions with highest density and place a server in every one. Computationally cheap.

O(N x max {log (N), K}).

Replica-Server Placement

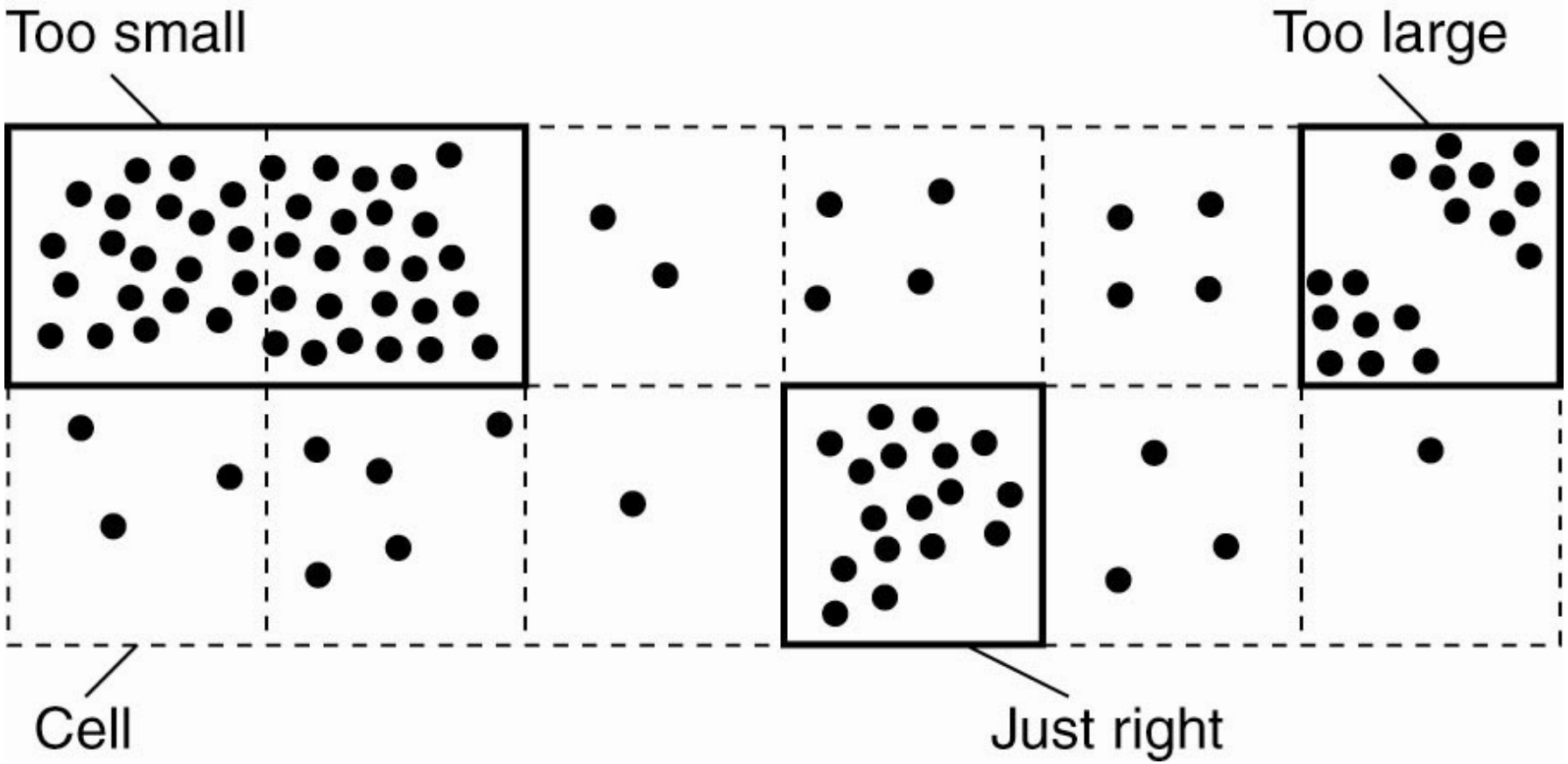


Figure 7-16. Choosing a proper cell size for server placement.

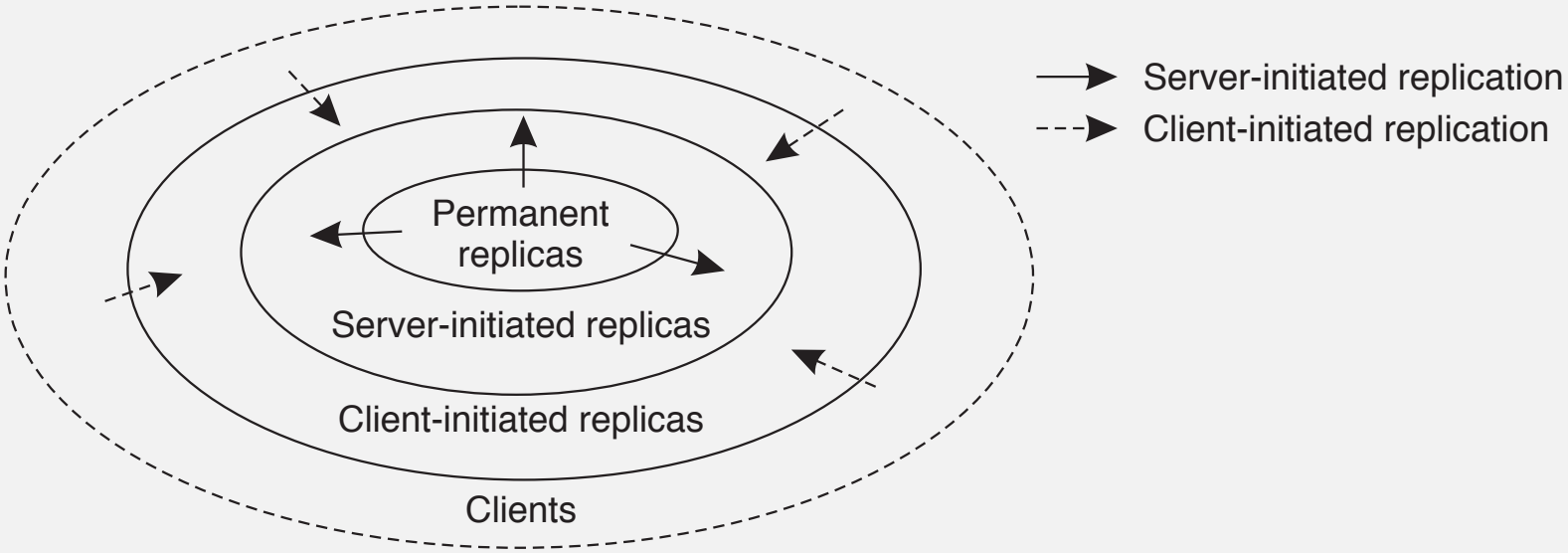
Content replication

Distinguish different processes

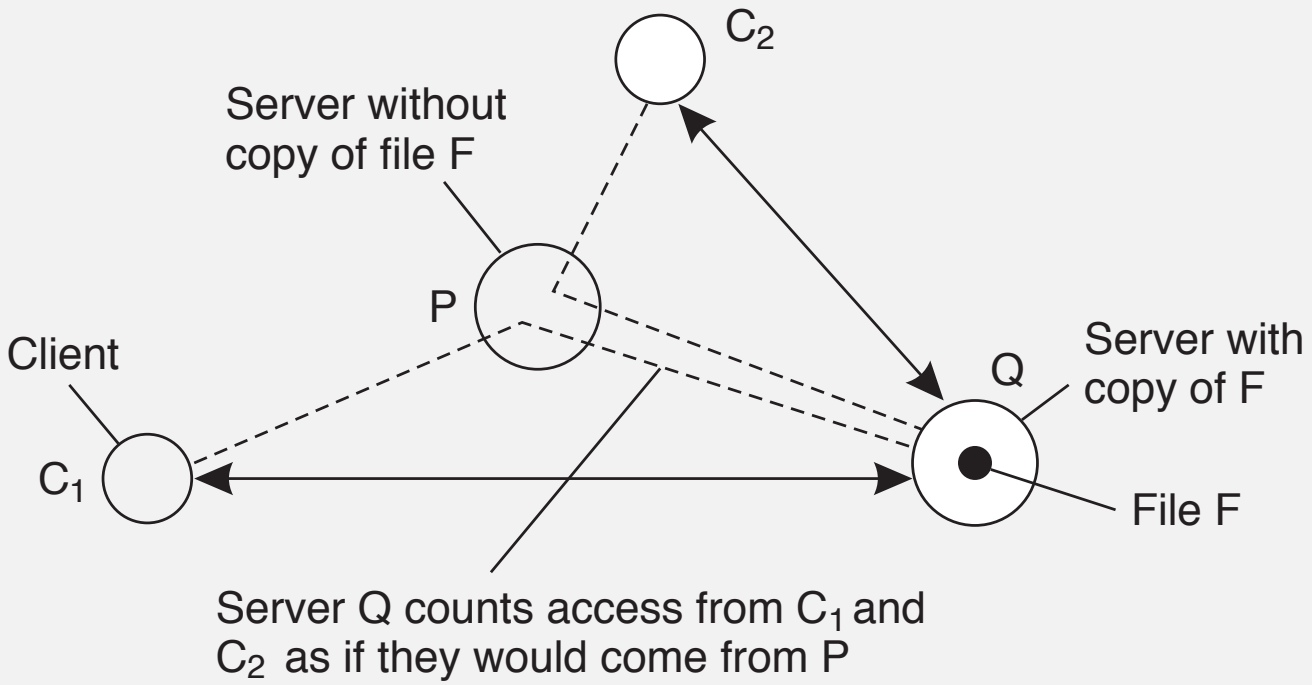
A process is capable of hosting a replica of an object or data:

- Permanent replicas: Process/machine always having a replica
- Server-initiated replica: Process that can dynamically host a replica on request of another server in the data store
- Client-initiated replica: Process that can dynamically host a replica on request of a client (client cache)

Content replication



Server-initiated replicas



- Keep track of access counts per file, aggregated by considering server closest to requesting clients
- Number of accesses drops below threshold $D \Rightarrow$ drop file
- Number of accesses exceeds threshold $R \Rightarrow$ replicate file
- Number of access between D and $R \Rightarrow$ migrate file

Content distribution

Client Initiated Replicas

Model

Consider only a client-server combination:

- Propagate only notification/invalidation of update (often used for caches)
- Transfer data from one copy to another (distributed databases): passive replication
- Propagate the update operation to other copies: active replication

Note

No single approach is the best, but depends highly on available bandwidth and read-to-write ratio at replicas.

Content distribution: client/server system

- Pushing updates: server-initiated approach, in which update is propagated regardless whether target asked for it.
- Pulling updates: client-initiated approach, in which client requests to be updated.

Issue	Push-based	Pull-based
1:	List of client caches	None
2:	Update (and possibly fetch update)	Poll and update
3:	Immediate (or fetch-update time)	Fetch-update time

1: State at server

2: Messages to be exchanged

3: Response time at the client

Content distribution

Observation

We can dynamically switch between pulling and pushing using leases: A contract in which the server promises to push updates to the client until the lease expires.

Content distribution

Issue

Make lease expiration time dependent on system's behavior (adaptive leases):

- Age-based leases: An object that hasn't changed for a long time, will not change in the near future, so provide a long-lasting lease
- Renewal-frequency based leases: The more often a client requests a specific object, the longer the expiration time for that client (for that object) will be
- State-based leases: The more loaded a server is, the shorter the expiration times become

Question

Why are we doing all this?

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 11, 2012

Dependability

- A component provides services to clients. To provide services, the component may require the services from other components → a component may depend on some other component.
- A component C depends on C* if the correctness of C's behavior depends on the correctness of C*'s behavior.
- Note: in the context of distributed systems, components are generally processes or channels.

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

Reliability versus Availability

- Reliability $R(t)$: probability that a component has been up and running continuously in the time interval $[0,t)$.
- Some traditional metrics:
  - Mean Time To Failure (MTTF): Average time until a component fails.
  - Mean Time To Repair (MTTR): Average time it takes to repair a failed component.
  - Mean Time Between Failures (MTBF): MTTF + MTTR

Reliability versus Availability

- Availability $A(t)$: Average fraction of time that a component has been up and running in the interval [0,t)
  - (Long term) availability A: $A(\infty)$
- Note:
  - A = MTTF/MTBF = MTTF/(MTTF + MTTR)

Observation

Reliability and availability make sense only if we have an accurate notion of what a failure actually is

Terminology

Term	Description	Example
Failure	May occur when a component is not living up to its specifications	A crashed program
Error	Part of a component that may lead to a failure	A programming bug
Fault	The cause of an error	A sloppy programmer

Terminology

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

Failure models

- Crash failures: Halt, but correct behavior until halting
- General omission failures: failure in sending or receiving messages
  - Receiving omissions: sent messages are not received
  - Send omissions: messages are not sent that should have
- Timing failures: correct output, but provided outside a specified time interval.
  - Performance failures: the component is too slow
- Response failures: incorrect output, but cannot be accounted to another component
  - Value failures: wrong output values
  - State transition failures: deviation from correct flow of control (Note: this failure may initially not even be observable)
- Arbitrary failures: any (combination of) failure may occur, perhaps even unnoticed

Dependability versus security

- Omission failure: A component fails to take an action that it should have taken
- Commission failure: A component takes an action that it should not have taken

Observations
Deliberate failures, be they omission or commission failures, stretch out to the field of security
There may actually be a thin line between dependability and security

Halting failures

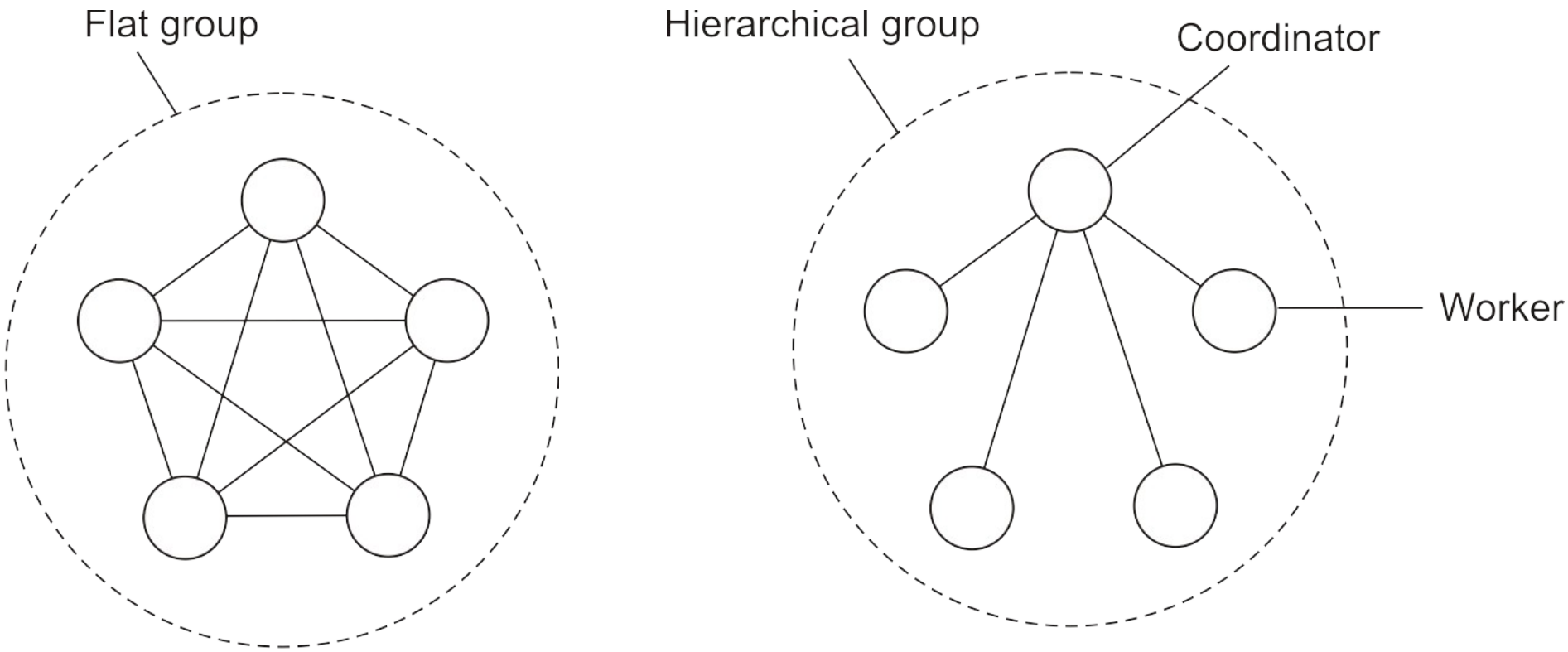
- Scenario: C no longer perceives any activity from C* — a halting failure? Distinguishing between a crash or omission/timing failure may be impossible:
  - Asynchronous system: no assumptions about process execution speeds or message delivery times → cannot reliably detect crash failures.
  - Synchronous system: process execution speeds and message delivery times are bounded → we can reliably detect omission and timing failures.
  - In practice we have partially synchronous systems: most of the time, we can assume the system to be synchronous, yet there is no bound on the time that a system is asynchronous → can normally reliably detect crash failures.

Halting failures

- Assumptions we can make:
  - Fail-stop: Crash failures, but reliably detectable
  - Fail-noisy: Crash failures, eventually reliably detectable
  - Fail-silent: Omission or crash failures: clients cannot tell what went wrong.
  - Fail-safe: Arbitrary, yet benign failures (can't do any harm).
  - Fail-arbitrary: Arbitrary, with malicious failures

Process reslience

- Basic idea: protect yourself against faulty processes through process replication:



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 must a k-fault-tolerant group be:
  - With halting failures (crash/omission/timing failures): we need *k+1* members: no member will produce an incorrect result, so the result of one member is good enough.
  - With arbitrary failures: we need *2k+1* members: the correct result can be obtained only through a majority vote.

Groups and failure masking

- Important:
  - All members are identical
  - All members process commands in the same order
- Result:
  - Only then do we know that all processes are programmed to do exactly the same thing.

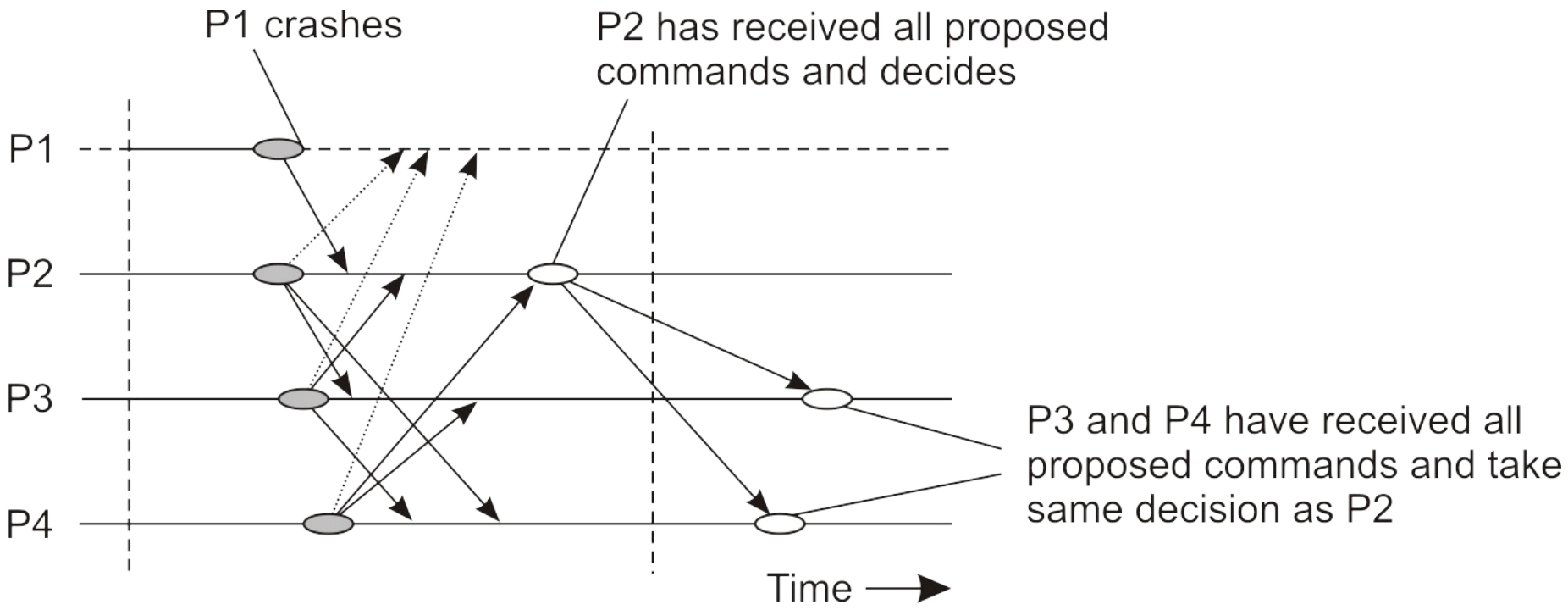
Observation

The processes need to have consensus on which command to execute next

Flooding-based consensus

- Assume:
  - Fail-crash semantics
  - Reliable failure detection
  - Unreliable communication
- Basic idea:
  - Processes multicast their proposed operations
  - All apply the same selection procedure → all process will execute the same if no failures occur
- Problem:
  - Suppose a process crashes before completing its multicast

Flooding-based consensus



Failure detection

Issue

How can we reliably detect that a process has actually crashed?

- General model:
  - Each process is equipped with a failure detection module
  - A process *p* probes another process *q* for a reaction
  - *q* reacts → *q* is alive
  - *q* does not react within *t* time units → *q* is suspected to have crashed
- Note: in a synchronous system:
  - a suspected crash is a known crash
  - Referred to as a perfect failure detector

Failure detection

- Practice: the eventually perfect failure detector
- Has two important properties:
  - Strong completeness: every crashed process is eventually suspected to have crashed by every correct process.
  - Eventual strong accuracy: eventually, no correct process is suspected by any other correct process to have crashed.
- Implementation:
  - If *p* did not receive heartbeat from *q* within time *t* → *p* suspects *q*.
  - If *q* later sends a message (received by *p*):
  - *p* stops suspecting *q*
  - *p* increases timeout value *t*
  - Note: if *q* does crash, *p* will keep suspecting *q.*

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

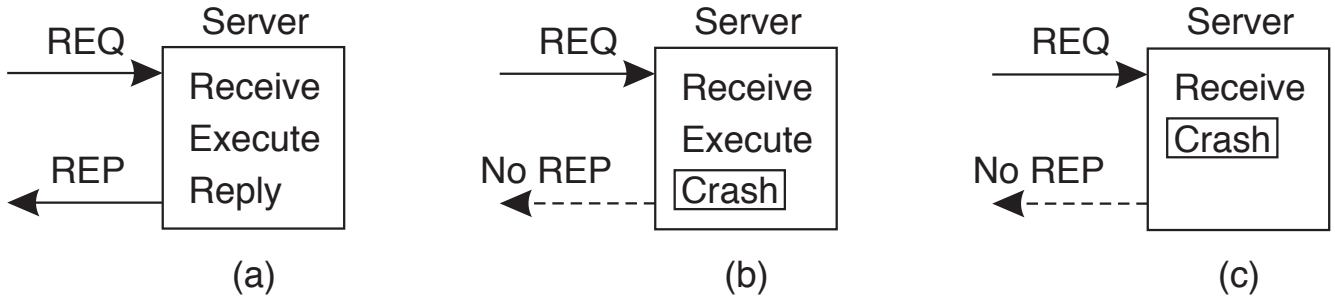
1: Relatively simple – just report back to client
2: Just resend message

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.

Client Print Request to Server Example:

Send the completion message (M), print the text (P), and crash (C).

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

OK = Text is printed once
DUP = Text is printed twice
ZERO = Text is not printed at all

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 $\Rightarrow$ servers kill orphans
- Require computations to complete in a T time units. Old ones are simply removed.

Question

What's the rolling back for?

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

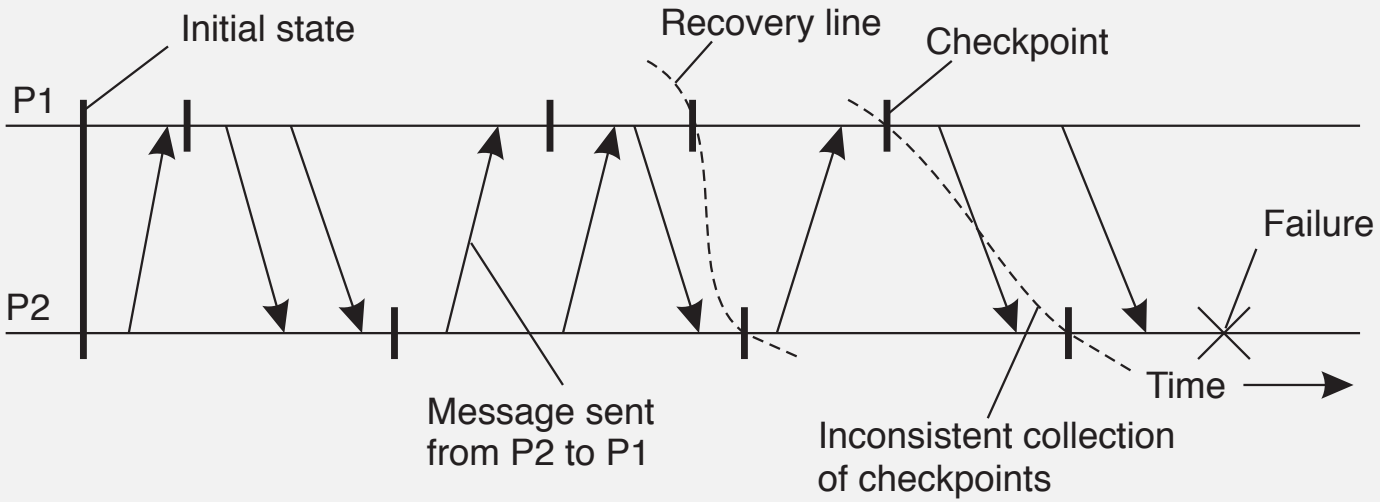
Consistent recovery state

Requirement

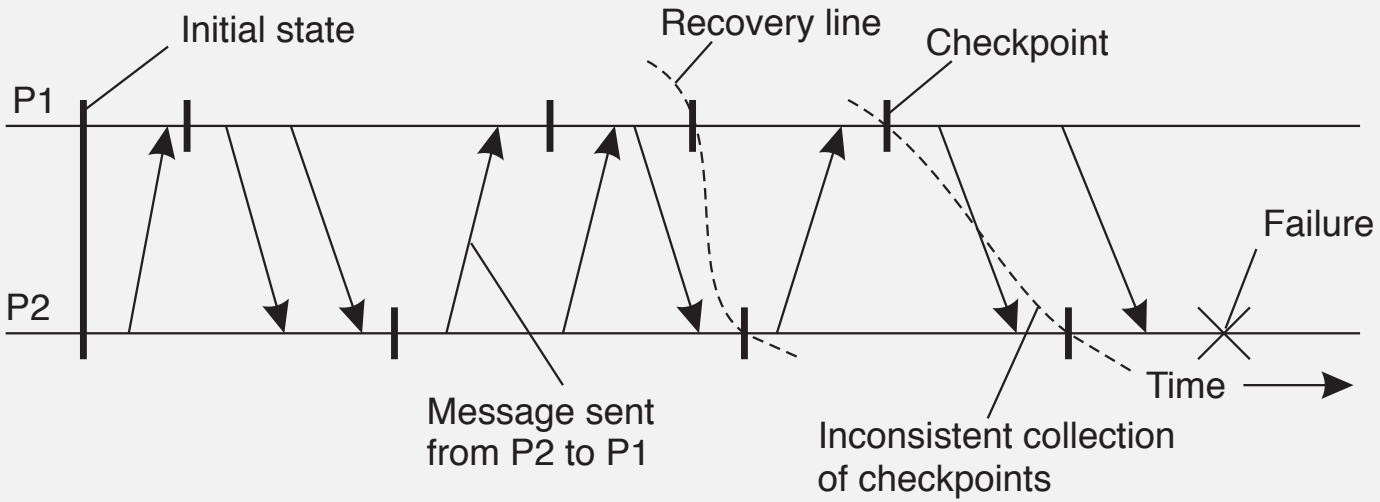
Every message that has been received is also shown to have been sent in the state of the sender.

Recovery line

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



Consistent recovery state



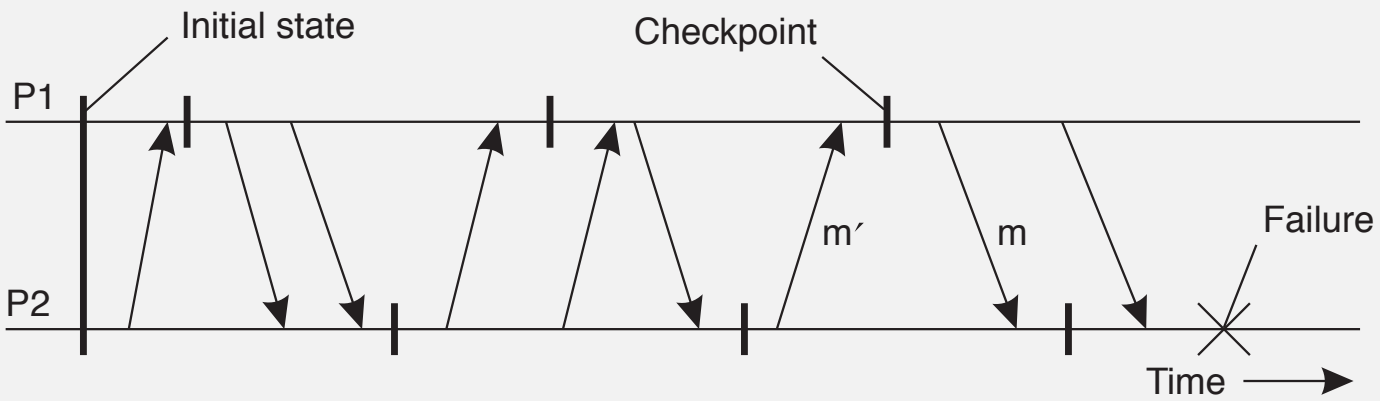
Observation

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



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