

Distributed Systems

Practical Byzantine Fault Tolerance

Alberto Montresor
Università di Trento

This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-sa/2.5/> or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.

Some of the slides written by Lorenzo Alvisi, Jinyang Li, Barbara Liskov

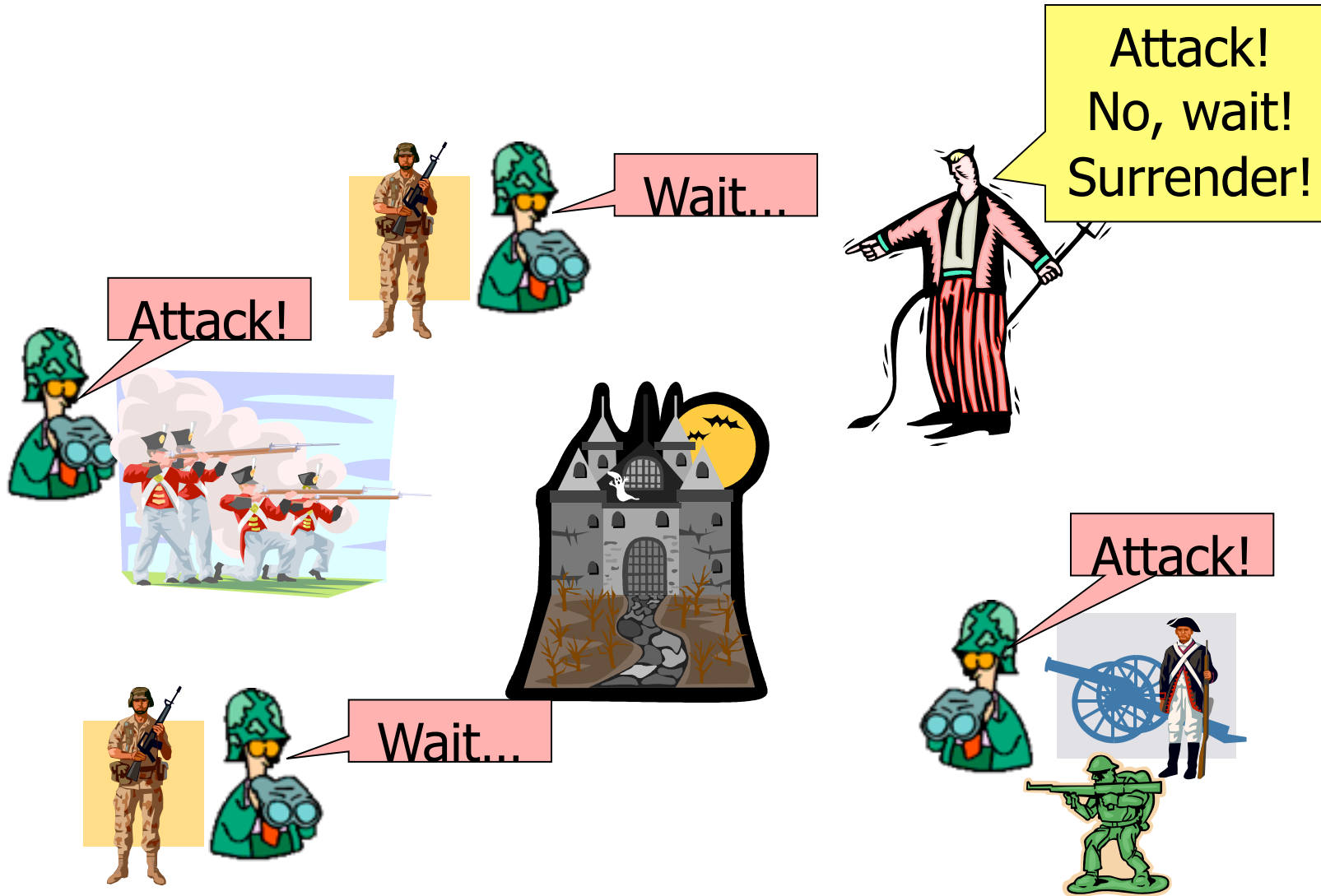
Motivation

- ◆ **Processes may exhibit arbitrary (byzantine) behavior**
 - ◆ Malicious attacks
 - ◆ They lie
 - ◆ They collude
 - ◆ Software error
 - ◆ Arbitrary states, messages
- ◆ **Example: Amazon outage (2008)**
 - ◆ “Root cause was a single bit flip in internal state messages”

Introduction

- ◆ **State-of-the-art at the end of the 90's**
 - ◆ Theoretically feasible algorithms to tolerate byzantine failures, but inefficient in practice
 - ◆ Assume synchrony – known bounds for message delays and processing speed
 - ◆ The most important: synchrony assumption needed for correctness – what about DoS?
- ◆ **Paper:**
 - ◆ L. Lamport, R. Shostak, M. Pease.
The Byzantine Generals Problem.
ACM TOPLAS, 4(3):382-401.

Byzantine generals



From cs4410 fall 08 lecture

Byzantine generals

- ◆ **A commanding general must send an order to his $n-1$ lieutenant generals such that**
 - ◆ IC1: All loyal lieutenants obey the same order.
 - ◆ IC2: If the commanding general is loyal, then every loyal lieutenant obeys the order he sends.
- ◆ **Assumptions**
 - ◆ Every message that is sent is received correctly
 - ◆ The receiver of a message knows who sent it
 - ◆ The absence of a message can be detected

“Oral Message” Algorithm $OM(m)$

◆ Algorithm $OM(0)$

1. The commander sends its value to every lieutenant
2. Each lieutenant uses the value he receives from commander, or uses RETREAT if he receives no value

◆ Algorithm $OM(m)$

1. The commander send its value to every lieutenant
2. $\forall i$, let v_i the value liut. i receives from comm, or RETREAT if no value. Liut. i acts as the commander of algorithm $OM(m-1)$ to send the value v_i to each of the other $n-2$ other lieutenants
3. $\forall i, j, i \neq j$, let v_j be the value received by liut. i from liut. j in step 2 of algorithm $OM(m-1)$ or RETREAT if no value. Liut i uses the value $majority(v_1, \dots, v_n)$ (deterministic function)

Byzantine generals problem

- ◆ **Theorem.**
 - ◆ For any m , Algorithm $OM(m)$ satisfies conditions IC1 and IC2 if there are more than $3m$ generals and at most m traitors
- ◆ **Problems:**
 - ◆ message paths of length up to $m+1$ (expensive)
 - ◆ absence of messages must be detected via time-out (vulnerable to DoS)

In the last ten years

- ◆ **A “byzantine renaissance”**

- ◆ Barbara Liskov and Miguel Castro.
Practical Byzantine Fault Tolerance.
In OSDI'99.

- ◆ **Contributions**

- ◆ First state machine replication protocol that survives Byzantine faults in asynchronous networks
- ◆ Live under weak byzantine assumptions – Byzantine paxos!
- ◆ Important optimizations
- ◆ Implementation of a Byzantine fault tolerant distributed file system (BFS)
- ◆ Experiments that measure cost of replication technique

The setup

- ◆ **System model**
 - ◆ Asynchronous distributed system
 - ◆ Unreliable channels
- ◆ **Unbreakable cryptography**
 - ◆ Public/private key pairs
 - ◆ MACs (message authentication codes)
 - ◆ Based on secret keys (client-server, server-server)
 - ◆ Collision-resistant hashes
- ◆ **Failure model**
 - ◆ Up to f byzantine servers
 - ◆ $N > 3f$ total servers
 - ◆ (Potentially byzantine clients)

The setup

- ◆ **Independent failures**
 - ◆ Different implementations of the service
 - ◆ Different operating systems
 - ◆ Different root passwords, different administrator

The specification

- ◆ **State machine replication**

- ◆ Replicated service with a state and deterministic operations operating on it
- ◆ Clients issue a request and block waiting for reply

- ◆ **Safety**

- ◆ The system satisfies linearizability, provided that $N > 3f + 1$
- ◆ Regardless of “faulty clients”... what?
 - ◆ all operations performed by faulty clients are observed in a consistent way by non-faulty clients
 - ◆ faulty clients cannot break invariants
- ◆ The algorithm does not rely on synchrony to provide safety...

The specification

- ◆ **Liveness**
 - ◆ It relies on synchrony to provide liveness
 - ◆ Assumes $delay(t)$ does not grow faster than t indefinitely
 - ◆ Weak assumption – if network faults are eventually repaired
 - ◆ Circumvent the impossibility results of FLP

Optimality

- ◆ $N=3f+1$ is optimal – why?
 - ◆ Max f processes are faulty
 - ◆ It must be possible to proceed after getting replies from at least $n-f$ replicas
 - ◆ since f replicas may be faulty and not responding
 - ◆ but it possible that the f replicas that not replied are not faulty, they are just slow
 - ◆ so f of the replicas which replied could be faulty
 - ◆ we are left with $n-2f$ replies
 - ◆ $n-2f$ must be larger than f (possibly colluding) replicas, so we have $n-2f > f \Leftrightarrow n > 3f$

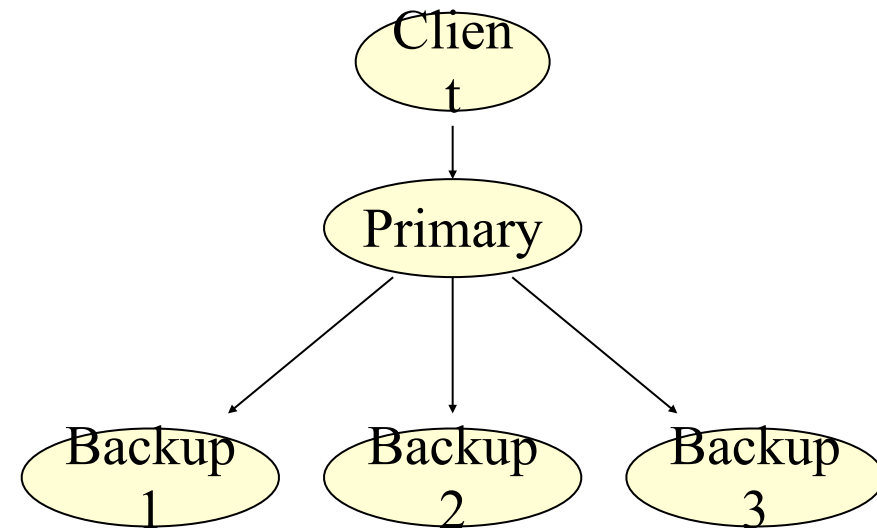
Replicas and views

- ◆ $N=3f+1$; more is useless
 - ◆ more and larger messages
 - ◆ without improving resiliency
- ◆ Replicas IDs: $0 \dots N-1$
- ◆ Replicas move through a sequence of configuration called **views**
 - ◆ **Primary** replica is i : $i = v \bmod N$
 - ◆ The other are **backups**
- ◆ **View changes** are carried out when the primary appears to have failed

The general idea

◆ The algorithm

- ◆ The client sends a request to invoke an operation on the primary
- ◆ The primary multicasts the request to the backups
- ◆ Quorums are employed to guarantee ordering on operations
- ◆ When an order has been agreed, replicas execute the request and send a reply to the client
- ◆ If the client receives at least $f+1$ identical replies, it is satisfied

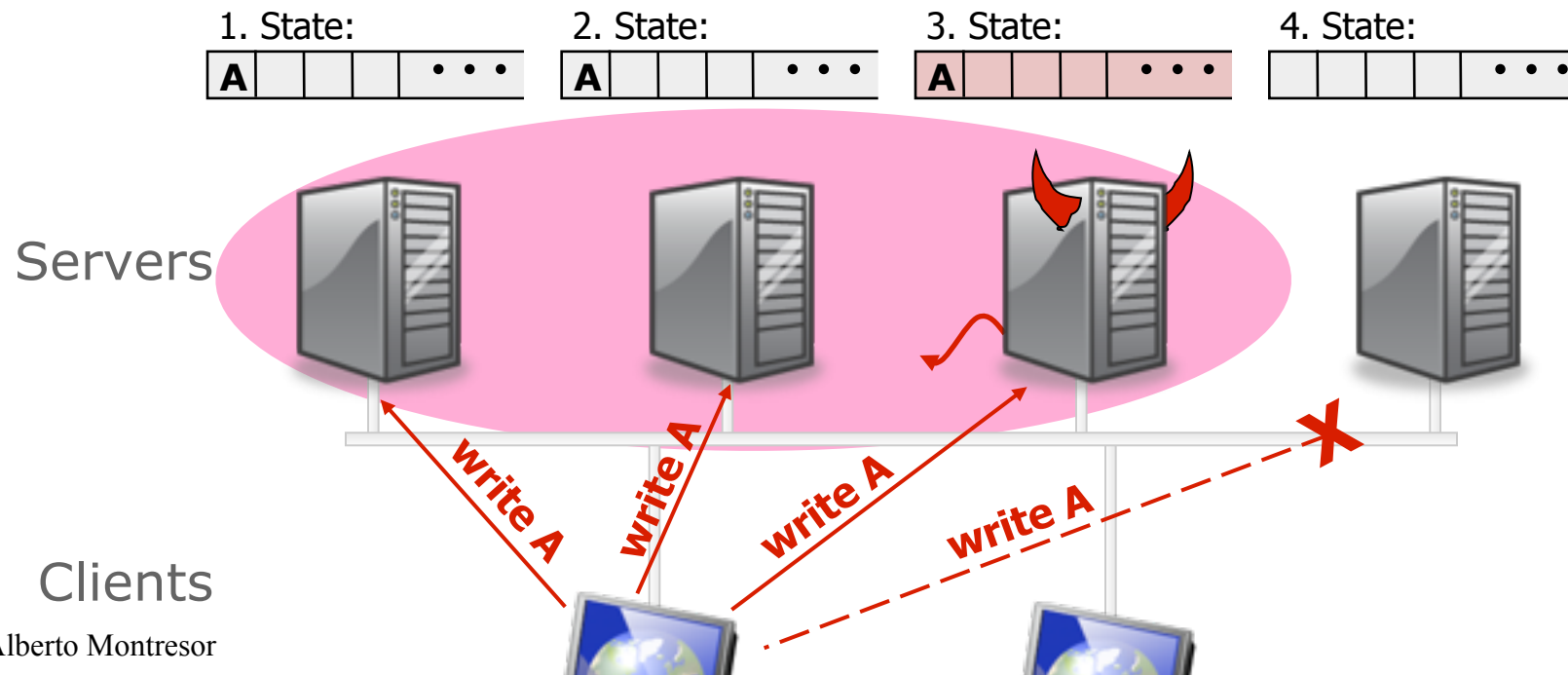


Problems

- ◆ **The Primary could be faulty!**
 - ◆ could ignore commands; assign same sequence number to different requests; skip sequence numbers; etc
 - ◆ Backups monitor primary's behavior and trigger view changes to replace faulty primary
- ◆ **Backups could be faulty!**
 - ◆ could incorrectly store commands forwarded by a correct primary
 - ◆ use dissemination Byzantine quorum systems [MR98]
- ◆ **Faulty replicas could incorrectly respond to the client!**
 - ◆ Client waits for $f + 1$ matching replies before accepting response

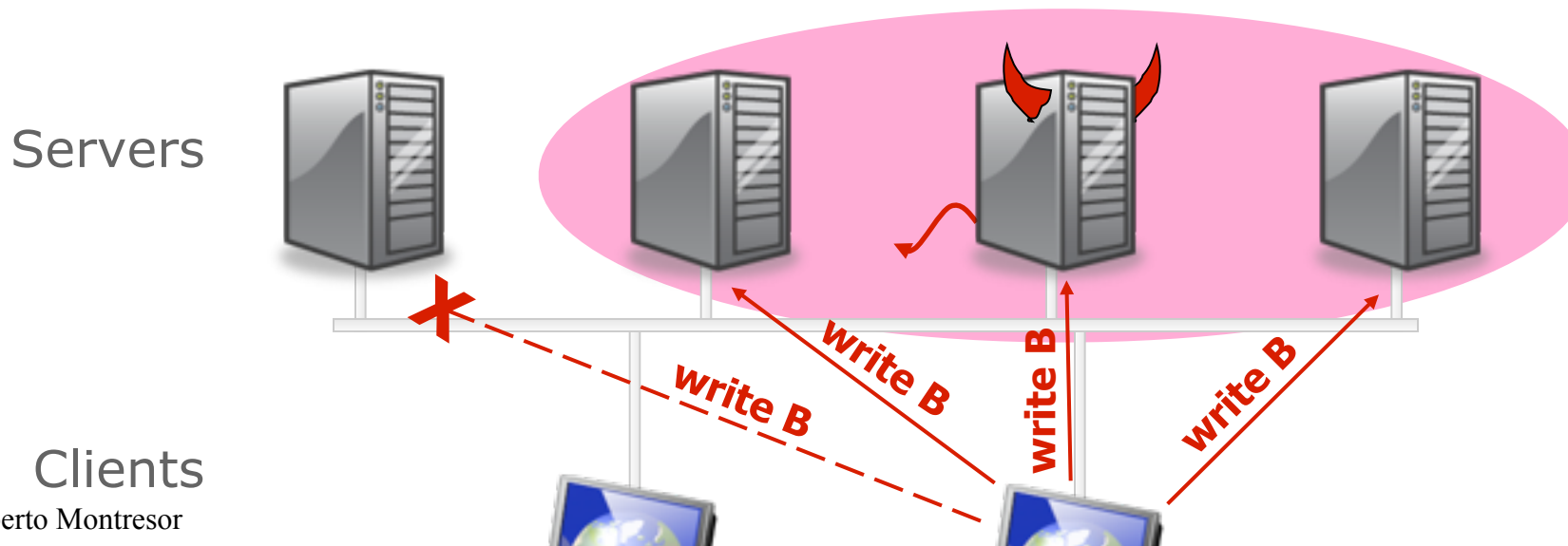
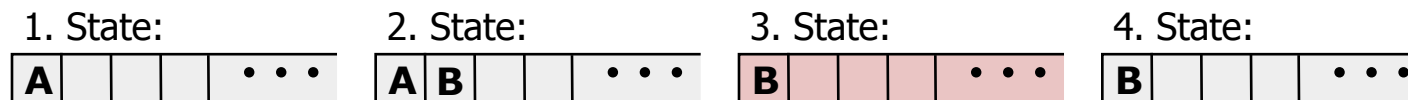
The general idea

- ◆ **Algorithm steps are justified by certificates**
 - ◆ Sets (quorums) of signed messages from distinct replicas proving that a property of interest holds
- ◆ **With quorums of size at least $2f + 1$**
 - ◆ Any two quorums intersect in at least one correct replica
 - ◆ Always one quorum that contains only non-faulty replicas



The general idea

- ◆ **Algorithm steps are justified by certificates**
 - ◆ Sets (quorums) of signed messages from distinct replicas proving that a property of interest holds
- ◆ **With quorums of size at least $2f + 1$**
 - ◆ Any two quorums intersect in at least one correct replica
 - ◆ There is always one quorum that contains only non-faulty replicas



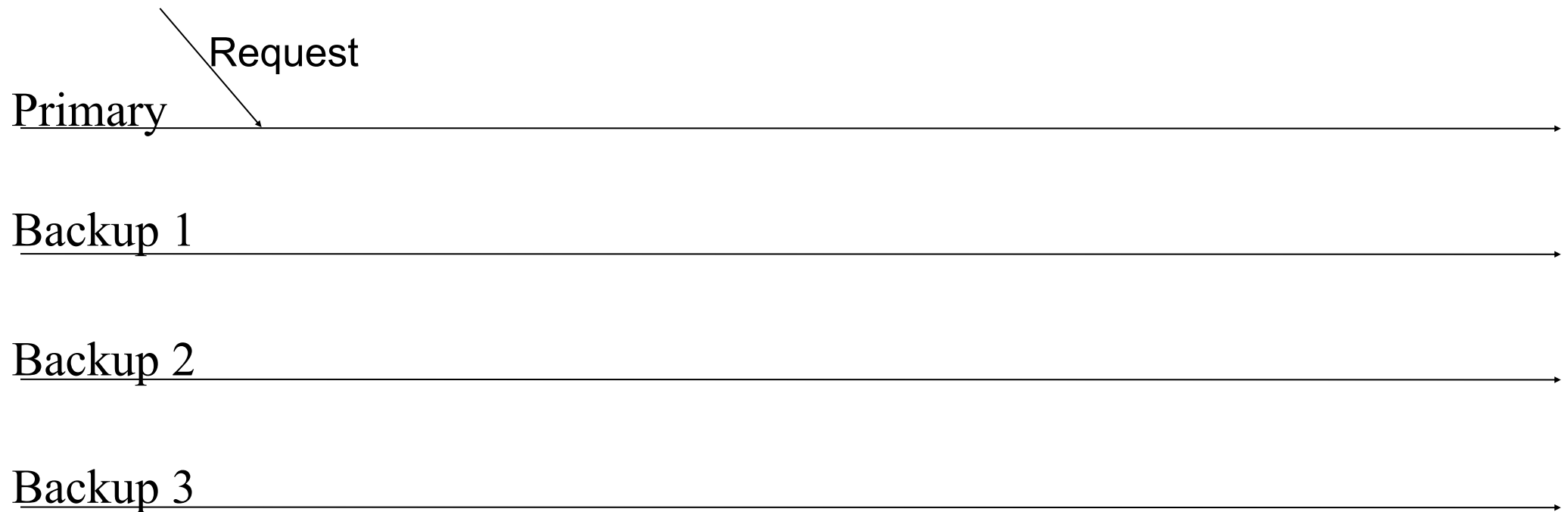
Protocol schema

- ◆ **Normal operation**
 - ◆ How the protocol works in the absence of failures
 - ◆ hopefully, the common case
- ◆ **View changes**
 - ◆ How to depose a faulty primary and elect a new one
- ◆ **Garbage collection**
 - ◆ How to reclaim the storage used to keep certificates
- ◆ **Recovery**
 - ◆ How to make a faulty replica behave correctly again
 - ◆ (not here)

State

- ◆ **The state of each replica includes**
 - ◆ the state of the actual service
 - ◆ a message log containing messages the replica has accepted
 - ◆ an integer denoting the replica current view

Client request



- ◆ $\langle \text{REQUEST}, o, t, c \rangle_{\sigma(c)}$
 - ◆ o : state machine operation
 - ◆ t : timestamp (used to ensure *exactly-once* semantics)
 - ◆ c : client id
 - ◆ $\sigma(c)$: client signature

Pre-prepare phase



◆ $\langle \langle \text{PRE-PREPARE}, v, n, d \rangle_{\sigma(p)}, m \rangle$

◆ v : current view

m : client message

◆ n : sequence number

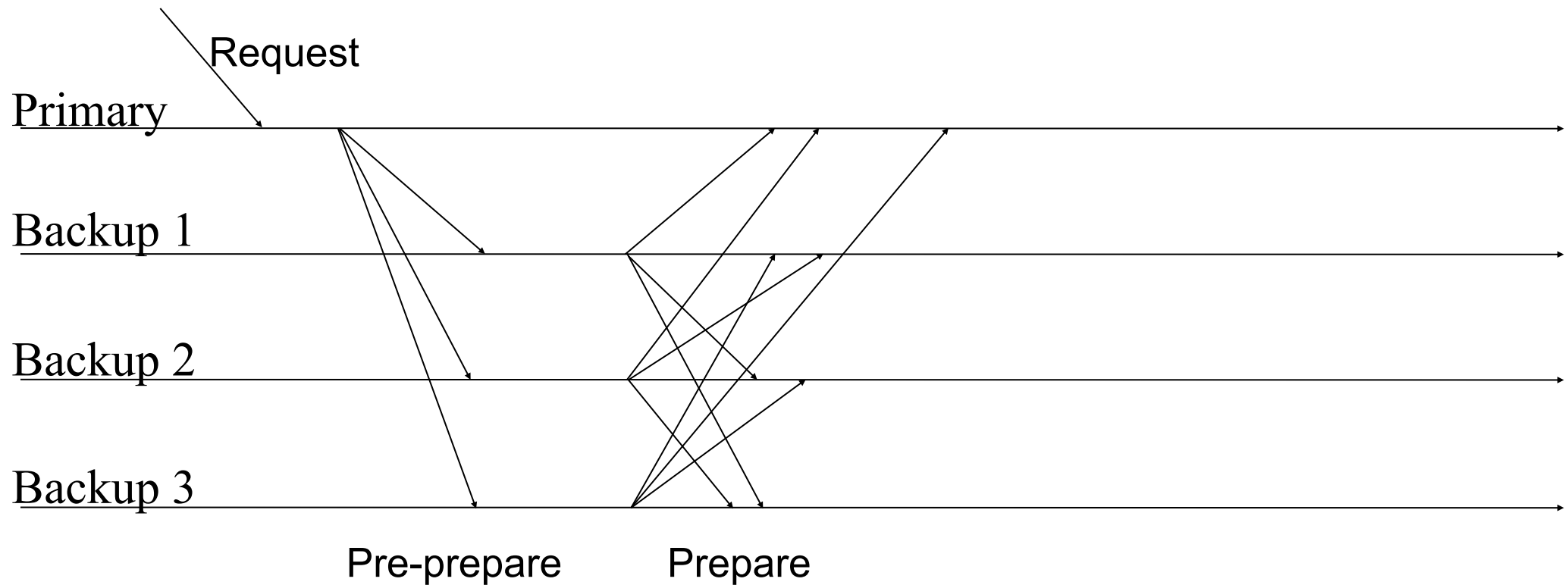
$\sigma(p)$: primary signature

◆ d : digest of client msg

Pre-prepare phase

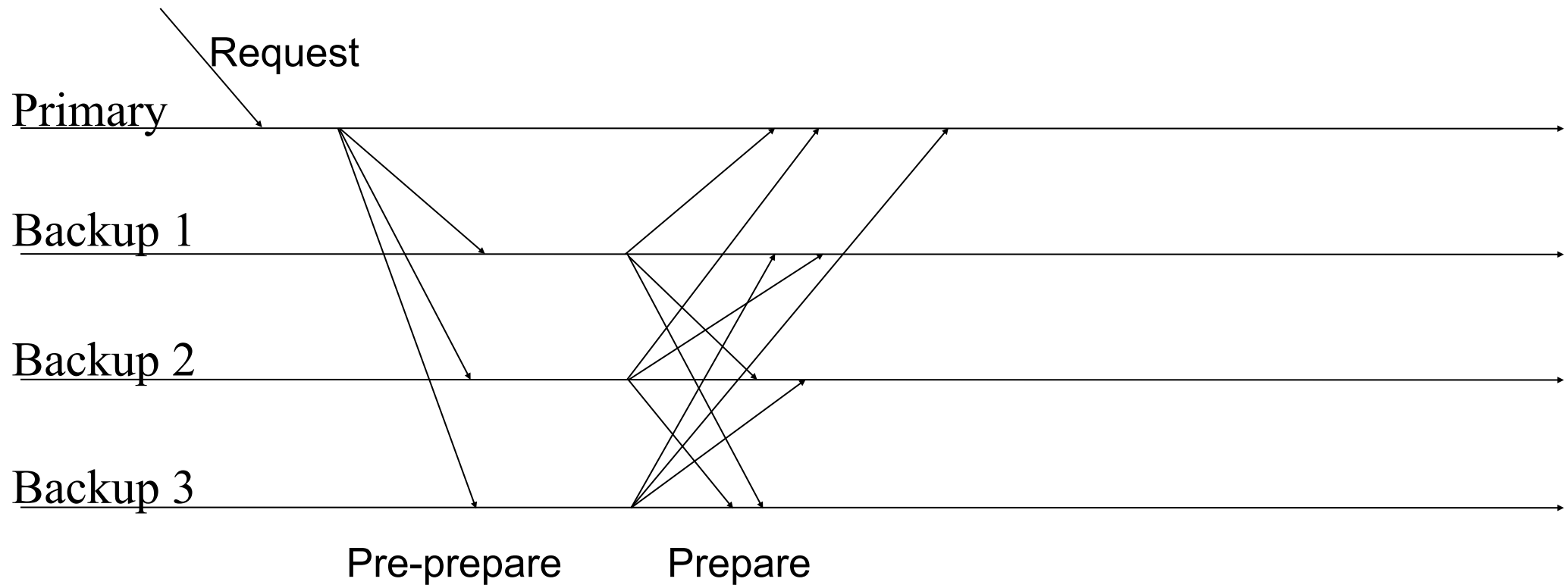
- ◆ $\langle \langle PRE-PREPARE, v, n, d \rangle_{\sigma(p)}, m \rangle$
- ◆ **Correct replica i accepts $PRE-PREPARE$ if**
 - ◆ the $PRE-PREPARE$ message is well formed
 - ◆ current view of i is v
 - ◆ i has not accepted another $PRE-PREPARE$ for v, n with a different d
 - ◆ n is between two water-marks L and H
(to avoid sequence number exhaustion caused by faulty primaries)
- ◆ **Each accepted $PRE-PREPARE$ message is stored in the accepting replica's message log (including the primary's)**

Prepare phase



- ◆ $\langle \text{PREPARE}, v, n, d \rangle_{\sigma(p)}$; *accepted by correct replicas if*
 - ◆ the *PREPARE* message is well formed
 - ◆ current view of *i* is *v*
 - ◆ *n* is between two water-marks *L* and *H*

Prepare phase



- ◆ $\langle PREPARE, v, n, d \rangle_{\sigma(i)}$
 - ◆ Replicas that send *PREPARE* accept seq.# n for m in view v
 - ◆ Each accepted *PREPARE* message is stored in the accepting replica's message log

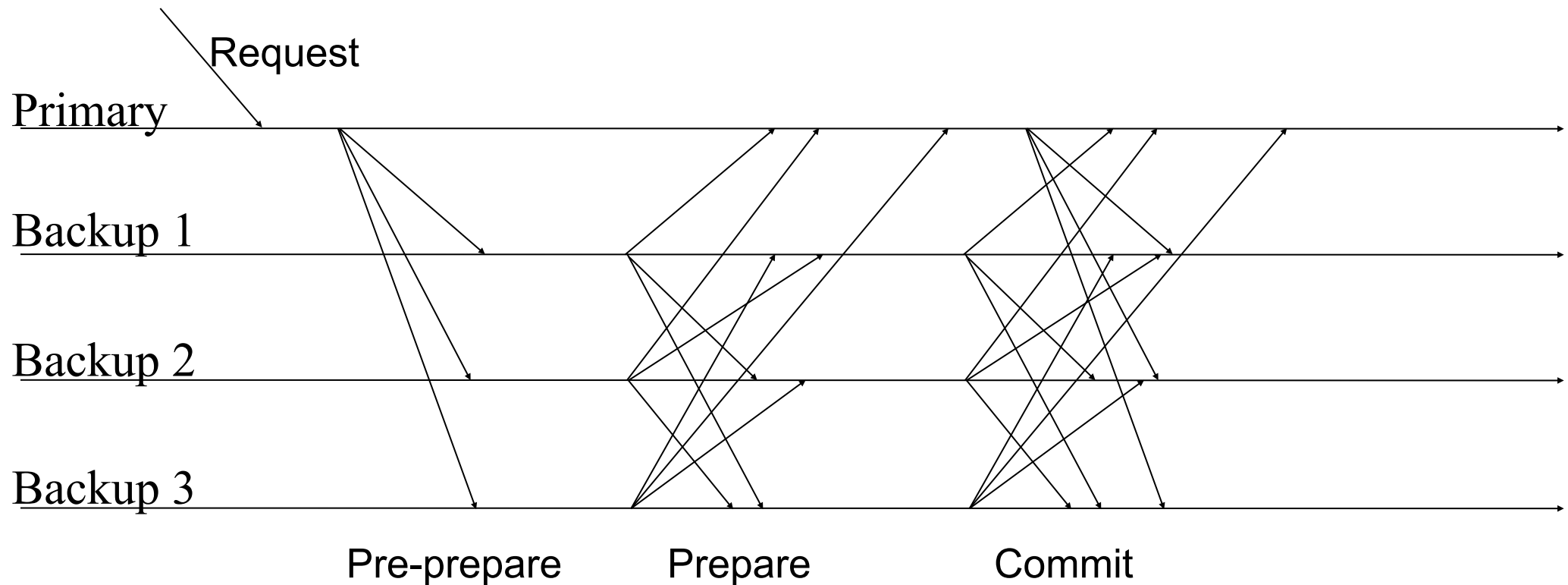
Prepare Certificate (P-certificate)

- ◆ **Replica i produces certificate $prepared(m,v,n,i)$ iff its log holds:**
 - ◆ The request m
 - ◆ A PRE-PREPARE for m in view v with sequence number n
 - ◆ $2f$ PREPARE from different backups that match the pre-prepare
- ◆ **$prepared(m,v,n,i)$ means that a quorum ($2f+1$) agrees with assigning sequence number n to m in view v**
- ◆ **Theorem: NO two non-faulty replicas i,j with**
 - ◆ $prepared(m, v, n, i)$, and $prepared(m',v, n, j)$
 - ◆ Proof?

Prepare Certificate

- ◆ **P-certificates ensure total order within views...**
- ◆ **but it is not enough:**
 - ◆ A P-certificate proves that a majority of correct replicas has agreed on a sequence number for a client's request
 - ◆ Yet that order could be modified by a new leader elected in a *view change*

Commit phase

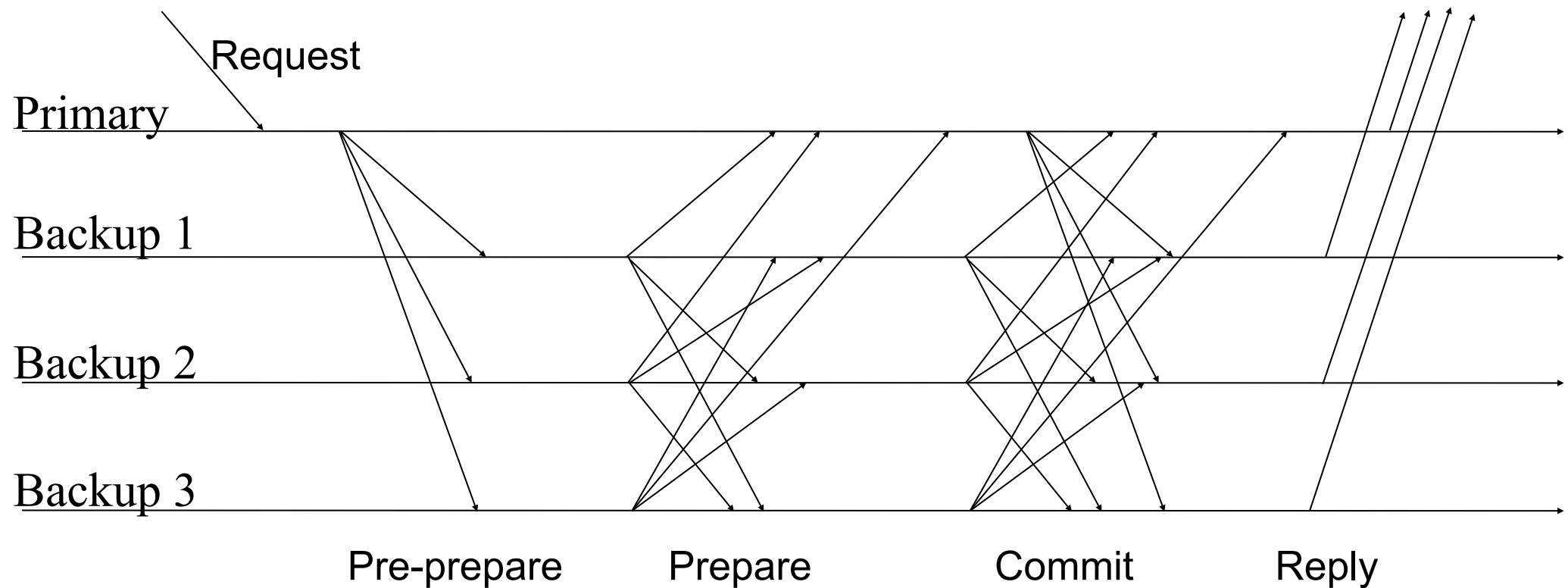


- ◆ After having collected a P-certificate $prepare(m, v, n, i)$, replica i sends $\langle COMMIT, v, n, d, i \rangle_{\sigma(i)}$; accepted if
 - ◆ the PREPARE message is well formed
 - ◆ current view of i is v
 - ◆ n is between two water-marks L and H

Commit Certificate (C-certificate)

- ◆ **Commit certificates ensure total order across views**
 - ◆ we guarantee that we can't miss prepare certificates during a view change
- ◆ **A replica has a certificate *committed*(m, v, n, i) if:**
 - ◆ it had a P-certificate *prepared*(m, v, n, i)
 - ◆ log contains $2f + 1$ matching *COMMIT* from different replicas (possibly including its own)
- ◆ **Replica executes a request after it gets commit certificate for it, and has cleared all requests with smaller sequence numbers**

Reply phase



- ◆ $\langle \text{REPLY}, v, t, c, i, r \rangle_{\sigma(i)}$
 - ◆ r is reply
 - ◆ $f+1$ replies are with the same t, r
 - ◆ If the client does not receive replies soon enough, it broadcast the request to all replicas

View changes

- ◆ **A un-satisfied replica backup i mutinies**
 - ◆ stops accepting messages (except VIEW-CHANGE and NEW-VIEW)
 - ◆ multicasts $\langle \text{VIEW-CHANGE}, v+1, P, i \rangle_{\sigma(i)}$
 - ◆ P contains a P-certificate P_m for each request m (up to a given number, see garbage collection)
- ◆ **Mutiny succeeds if new primary collects a *new-view certificate* V ,**
 - ◆ $2f+1$ VIEW-CHANGE messages
 - ◆ indicating that $2f + 1$ distinct replicas (including itself) support the change of leadership

View change

- ◆ The “primary elect” p' (replica $v+1 \bmod N$)
 - ◆ p' extracts from the new-view certificate V :
 - ◆ the highest sequence number h of any message for which V contains a P-certificate
 - ◆ a new PRE-PREPARE message for any message with sequence number $n \leq h$, in either set O
 - ◆ If there is a P-certificate for n, d in V $d = \text{Digest}(m)$
 - ◆ $O = O \cup \langle \text{PRE-PREPARE}, v+1, n, d \rangle_{\sigma(p')}$
 - ◆ Otherwise
 - ◆ $O = O \cup \langle \text{PRE-PREPARE}, v+1, n, d^{\text{null}} \rangle_{\sigma(p')}$
 - ◆ p' multicasts $\langle \text{NEW-VIEW}, v+1, V, O \rangle_{\sigma(p')}$

View change

- ◆ **Backup accepts NEW-VIEW message for $v+1$ if**
 - ◆ it is signed properly by p'
 - ◆ it contains in V valid VIEW-CHANGE messages for $v+1$
 - ◆ it can verify locally that O is correct (repeating the primary's computation)
- ◆ **Actions:**
 - ◆ Adds all entries in O to its log (so did p')
 - ◆ Multicasts a PREPARE for each message in O
 - ◆ Adds all PREPARE to log and enters new view

Garbage Collection

- ◆ **A correct replica keeps in log messages about request o until:**
 - ◆ o has been executed by a majority of correct replicas, and
 - ◆ this fact can be proven during a view change
- ◆ **Truncate log with stable checkpoints**
 - ◆ Each replica i periodically (after processing k requests) checkpoints state and multicasts $\langle \text{CHECKPOINT}, n, d, i \rangle$
 - ◆ n : last executed request
 - ◆ d : state digest
- ◆ **$2f + 1$ equal CHECKPOINT messages are a proof of the checkpoint's correctness (*stable checkpoint certificate*)**

View Change, revisited

- ◆ **Message** $\langle \text{VIEW-CHANGE}, v+1, n, s, C, P, i \rangle_{\sigma(i)}$
 - ◆ n : the seq# of last stable checkpoint
 - ◆ s : the last stable checkpoint
 - ◆ C : the checkpoint certificate ($2f+1$ checkpoint messages)
- ◆ **Message** $\langle \text{NEW-VIEW}, v+1, n, V, O \rangle_{\sigma(p')}$
 - ◆ n : the seq# of last stable checkpoint
 - ◆ V, O : contains only request with seq# larger than n

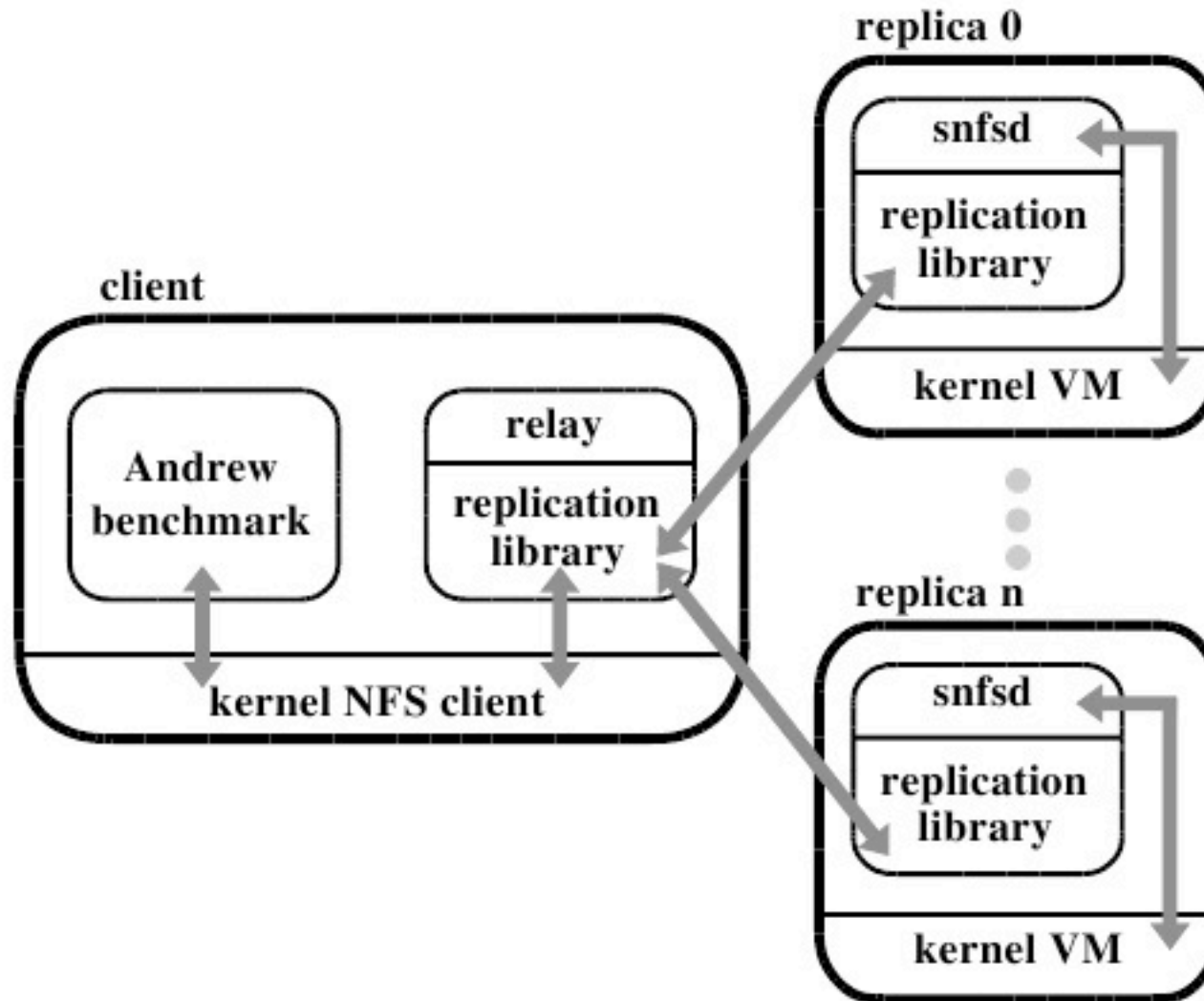
Optimization: reducing communication

- ◆ **Reducing replies**
 - ◆ One replica designated to send reply to client
 - ◆ Other replicas send digest of the reply
- ◆ **Lower latency for writes (4 messages)**
 - ◆ Replicas respond at prepare (*tentative execution*)
 - ◆ Client waits for $2f+1$ matching responses
- ◆ **Fast reads (one round trip)**
 - ◆ Client sends to all; they respond immediately
 - ◆ Client waits for $2f+1$ matching responses

Optimization: Cryptography

- ◆ **Reducing overhead**
 - ◆ Public-key cryptography only for view changes
 - ◆ MACs (message authentication codes) for all other messages
- ◆ **To give an idea (Pentium 200Mhz)**
 - ◆ Generating 1024-bit RSA signature of a MD5 digest: 43ms
 - ◆ Generating a MAC of the same message: 10 μ s

Implementation: Byzantine NFS server



Performance Evaluation

arg./res. (KB)	replicated		without replication
	read-write	read-only	
0/0	3.35 (309%)	1.62 (98%)	0.82
4/0	14.19 (207%)	6.98 (51%)	4.62
0/4	8.01 (72%)	5.94 (27%)	4.66

Table 1: Micro-benchmark results (in milliseconds); the percentage overhead is relative to the unreplicated case.

phase	BFS		BFS-nr
	strict	r/o lookup	
1	0.55 (57%)	0.47 (34%)	0.35
2	9.24 (82%)	7.91 (56%)	5.08
3	7.24 (18%)	6.45 (6%)	6.11
4	8.77 (18%)	7.87 (6%)	7.41
5	38.68 (20%)	38.38 (19%)	32.12
total	64.48 (26%)	61.07 (20%)	51.07

Table 2: Andrew benchmark: BFS vs BFS-nr. The times are in seconds.

phase	BFS		NFS-std
	strict	r/o lookup	
1	0.55 (-69%)	0.47 (-73%)	1.75
2	9.24 (-2%)	7.91 (-16%)	9.46
3	7.24 (35%)	6.45 (20%)	5.36
4	8.77 (32%)	7.87 (19%)	6.60
5	38.68 (-2%)	38.38 (-2%)	39.35
total	64.48 (3%)	61.07 (-2%)	62.52

Table 3: Andrew benchmark: BFS vs NFS-std. The times are in seconds.

Further work

◆ Papers

- ◆ R. Kotla, A. Clement, E. Wong, L. Alvisi and M. Dahlin.
Zyzyva: Speculative Byzantine Fault Tolerance.
In Proc. of the ACM Symposium on Operating Systems Principles (SOSP'07), Stevenson, WA, October 2007.
- ◆ A. Clement, M. Marchetti, E. Wong, L. Alvisi, and M. Dahlin
Making Byzantine Fault-Tolerant Systems Tolerate Byzantine Faults.
In Proc. of the 6th USENIX Symposium of Network Systems Design and Implementation (NSDI '09) , Boston, MA, April 2009.