Distributed Systems
Practical Byzantine Fault Tolerance

Alberto Montresor

University of Trento, Italy

November 30, 2012

Contents

1 Introduction
2 Byzantine generals
3 Practical Byzantine fault tolerance
4 Bibliography

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"

History

- 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?

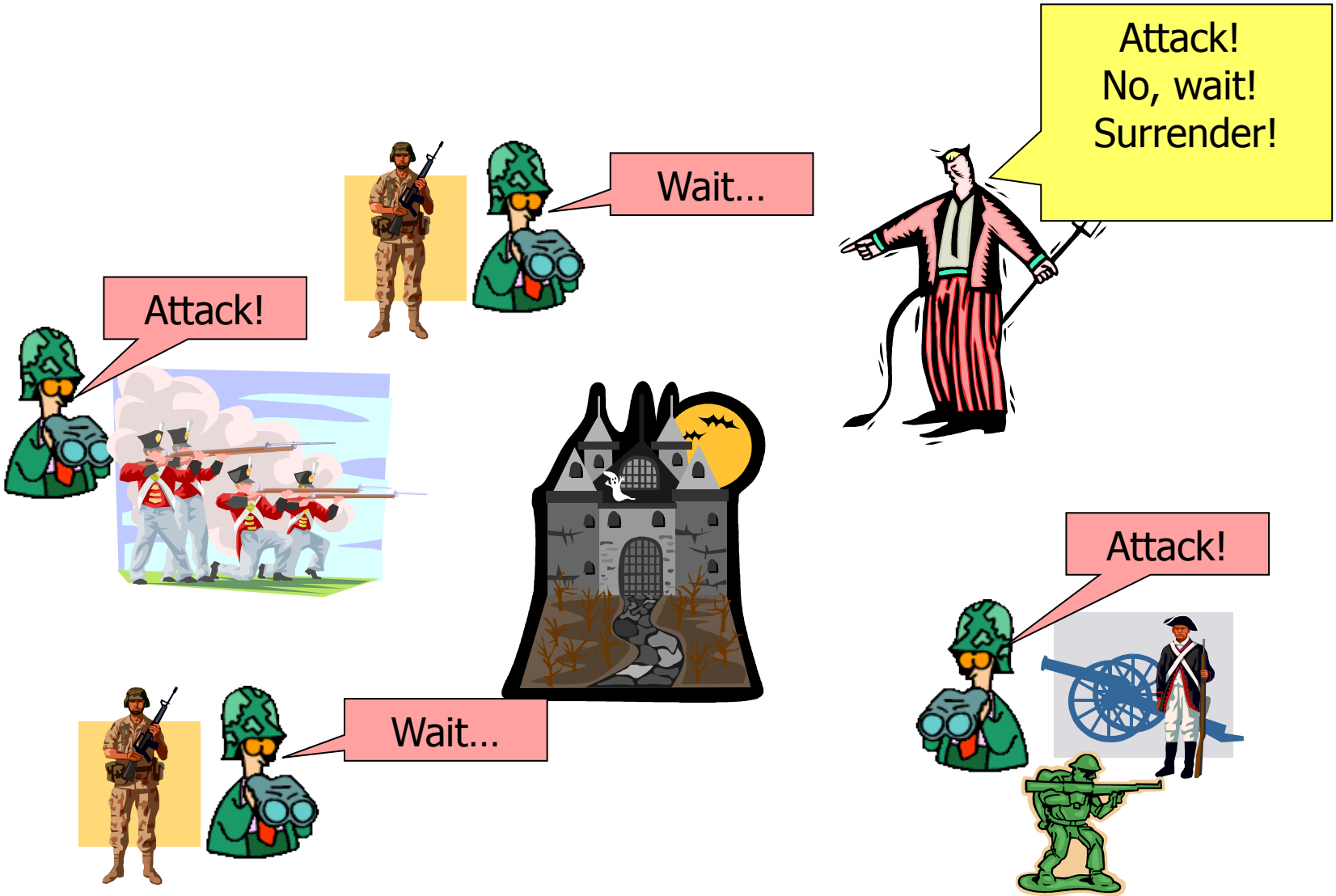
Relevant bibliography

L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982. `http://www.disi.unitn.it/~montreso/ds/papers/ByzantineGenerals.pdf`

Contents

1 Introduction
2 Byzantine generals
3 Practical Byzantine fault tolerance
4 Bibliography

Byzantine generals

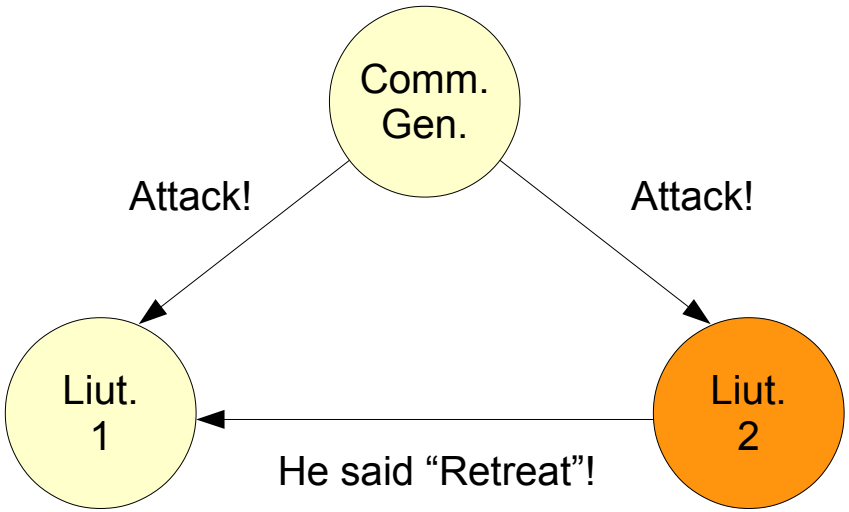


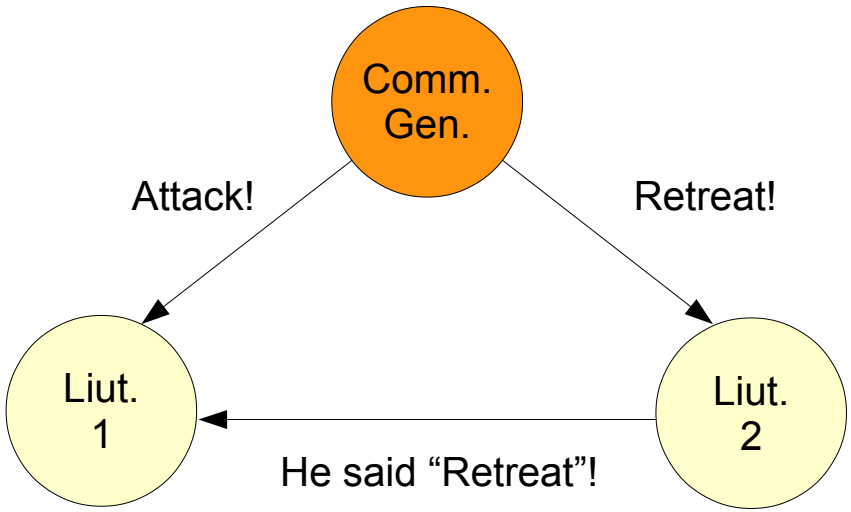
Specification

- 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

Impossibility results

- "Oral" messages
- No solution with three general can handle a single traitor

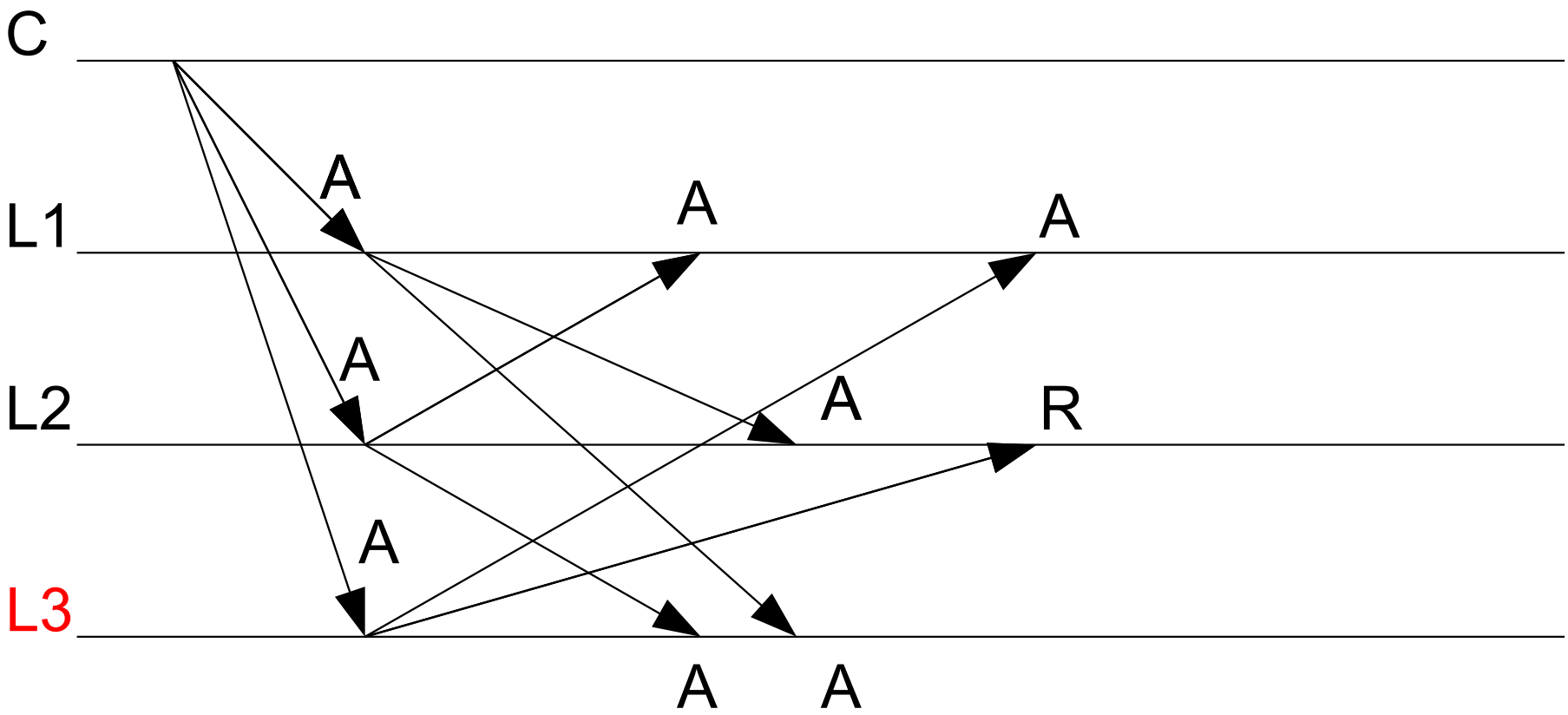




"Oral Message" algorithm $OM(m)$

- Algorithm $OM(0)$
  1. The commander sends its value to every lieutenant
  2. Each lieutenant returns the value he received from commander, or returns RETREAT if he received no value
- Algorithm $OM(m)$
  1. The commander send its value to every lieutenant
  2. $\forall i$, let v_i be the value lieutenant i receives from the commander, or RETREAT if it has received no value. Lieutenant 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 i from j in step 2 of algorithm $OM(m-1)$ or RETREAT if no value. Lieutenant i uses the value $\mathsf{majority}(v_1, ..., v_n)$ (deterministic function)

"Oral Message" Algorithm Example – $OM(1)$



Oral messages

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)

Contents

1 Introduction
2 Byzantine generals
3 Practical Byzantine fault tolerance
4 Bibliography

A Byzantine "renaissance"

Relevant bibliography

M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 20:398–461, Nov. 2002.
`http://www.disi.unitn.it/~montreso/ds/papers/PbftTocs.pdf`

Contributions

- First state machine replication protocol that survives Byzantine faults in asynchronous networks
- Live under weak Byzantine assumptions – Byzantine paxos!
- Implementation of a Byzantine, fault tolerant distributed FS
- Experiments measuring cost of replication technique

Assumptions

- System model
  - Asynchronous distributed system with N processes
  - Unreliable channels
- Unbreakable cryptography
  - Message m is signed by its sender i, and we write $\langle m \rangle_{\sigma(i)}$, through:
    - Public/private key pairs
    - Message authentication codes (MAC)
  - A digest $d(m)$ of message m is produced through collision-resistant hash functions

Assumptions

- Failure model
  - Up to f Byzantine servers
  - $N > 3f$ total servers
  - (Potentially Byzantine clients)
- Independent failures
  - Different implementations of the service
  - Different operating systems
  - Different root passwords, different administrator

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"...
    - ⋆ all operations performed by faulty clients are observed in a consistent way by non-faulty clients
  - The algorithm does not rely on synchrony to provide safety...
- 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

Theorem

To tolerate up to f malicious nodes, N must be equal to $3f + 1$

Proof.

Optimality

Theorem

To tolerate up to f malicious nodes, N must be equal to $3f + 1$

Proof.

- It must be possible to proceed after communicating with $N - f$ replicas, because the faulty replicas may not respond

Optimality

Theorem

To tolerate up to f malicious nodes, N must be equal to $3f + 1$

Proof.

- It must be possible to proceed after communicating with $N - f$ replicas, because the faulty replicas may not respond
- But the f replicas not responding may be just slow, so f of those that responded might be faulty

Optimality

Theorem

To tolerate up to f malicious nodes, N must be equal to $3f + 1$

Proof.

- It must be possible to proceed after communicating with $N - f$ replicas, because the faulty replicas may not respond
- But the f replicas not responding may be just slow, so f of those that responded might be faulty
- The correct replicas who responded $(N - 2f)$ must outnumber the faulty replicas, so

$$N - 2f > f \Rightarrow N > 3f$$

Optimality

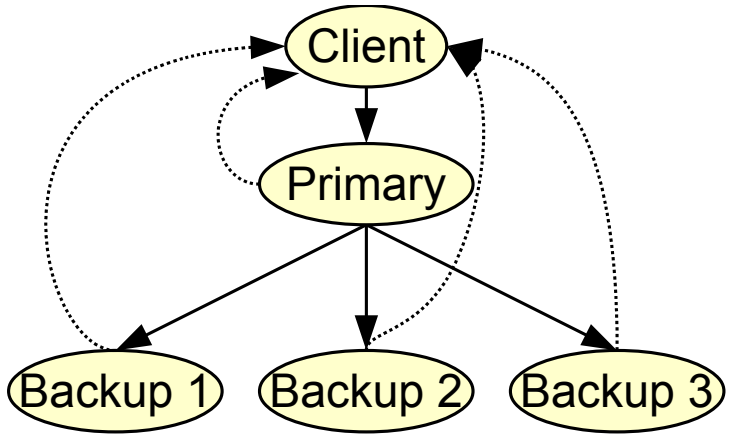
- So, $N > 3f$ to ensure that at least a correct replica is present in the reply set
- $N = 3f + 1$; more is useless
  - more and larger messages
  - without improving resiliency

Processes and views

- Replicas IDs: $0 \dots N-1$
- Replicas move through a sequence of configurations called views
- During view v:
  - 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 algorithm

- To invoke an operation, the client sends a request to 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
- When the client receives at least $f + 1$ identical replies, it is satisfied

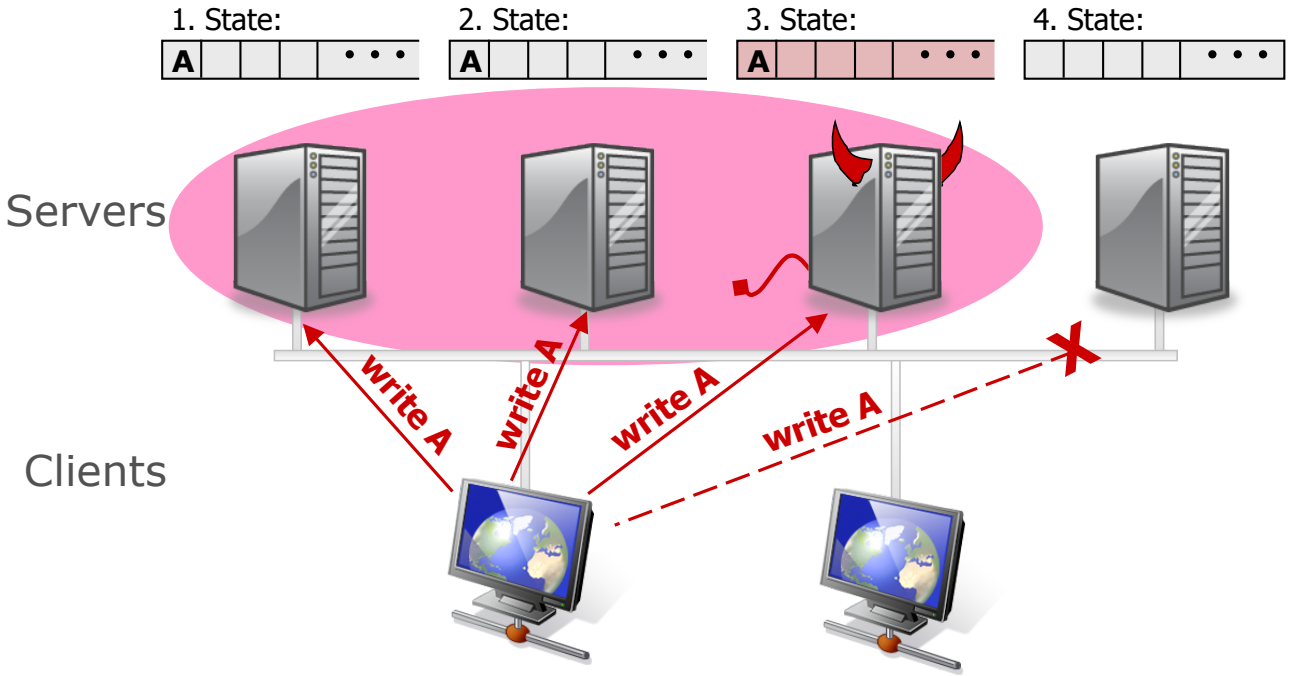


Problems

- 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
- Faulty replicas could incorrectly respond to the client!
  - Client waits for $f + 1$ matching replies before accepting response

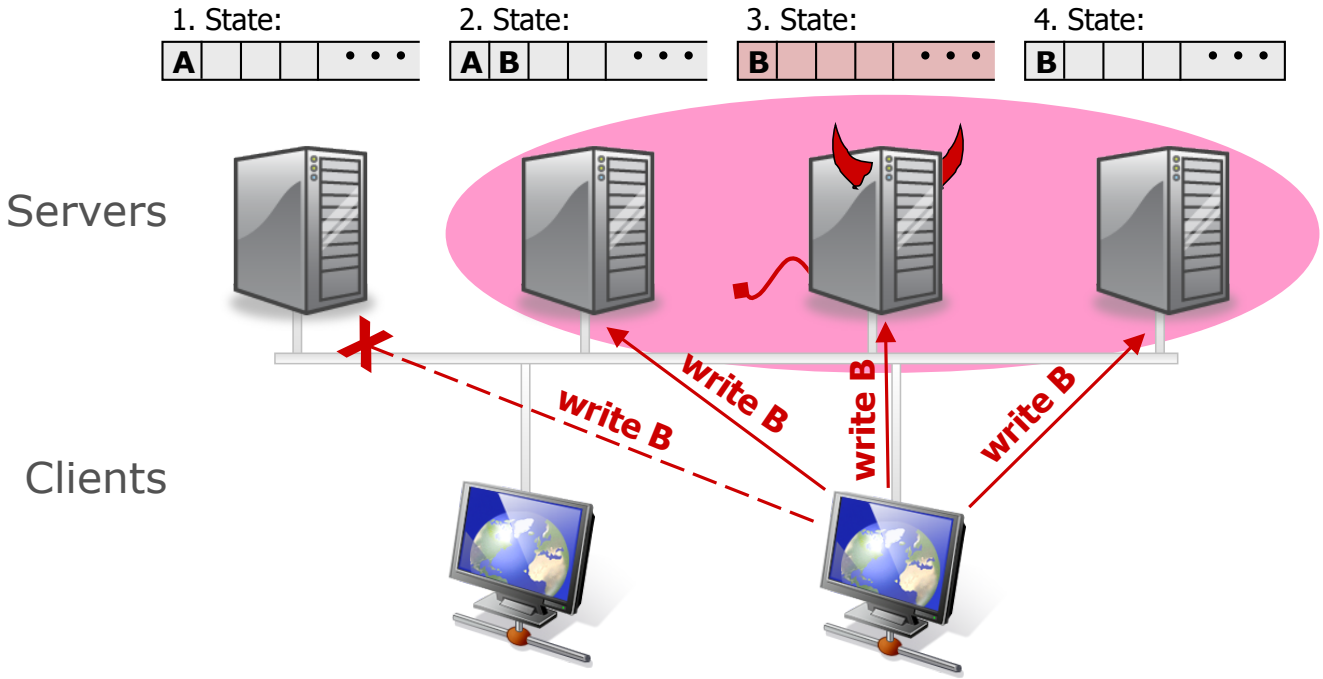
The general idea

- 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
  - There is 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 internal state of each of the replicas include:
  - the state of the actual service
  - a message log containing all the 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



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

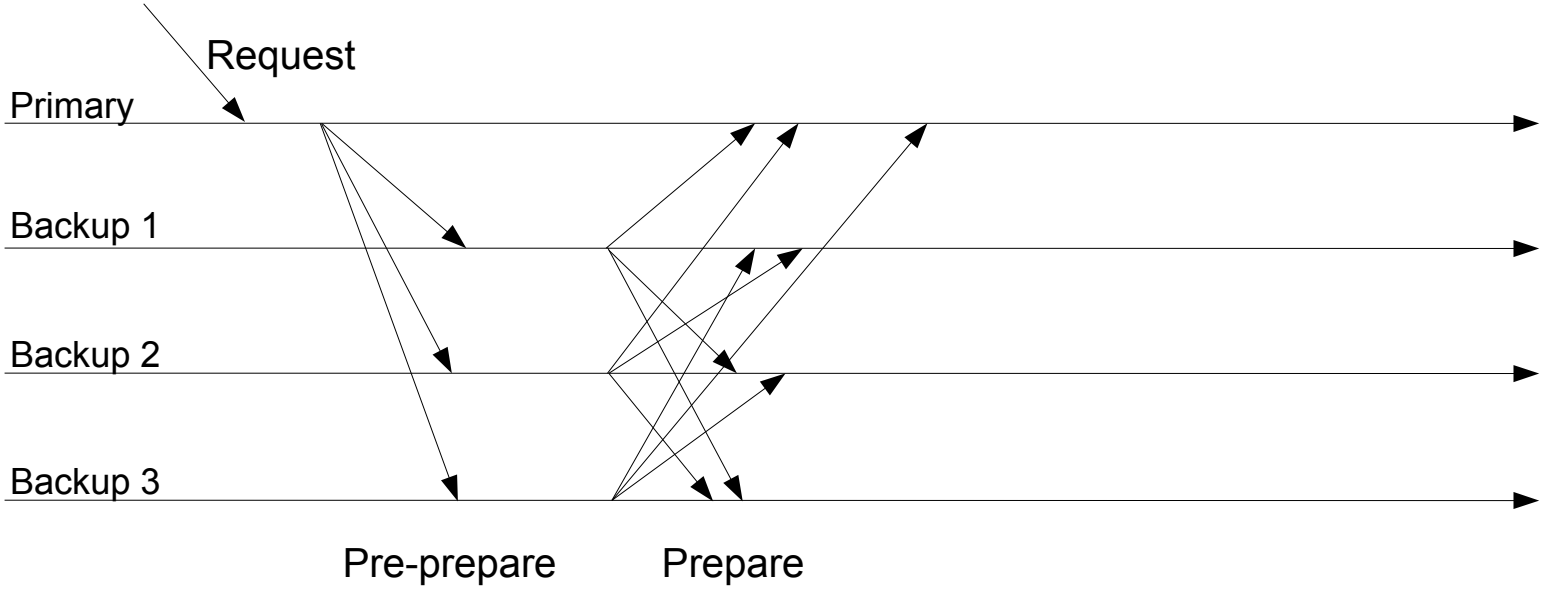
- v: current view
- n: sequence number
- $d(m)$: digest of client message
- $\sigma(p)$: primary signature
- m: client message

Pre-prepare phase

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

- Correct replica i accepts PRE-PREPARE if:
  - the PRE-PREPARE message is well formed
  - the current view of i is v
  - i has not accepted another PRE-PREPARE for v, n with a different digest
  - 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)
- Non-accepted PRE-PREPARE messages are just discarded

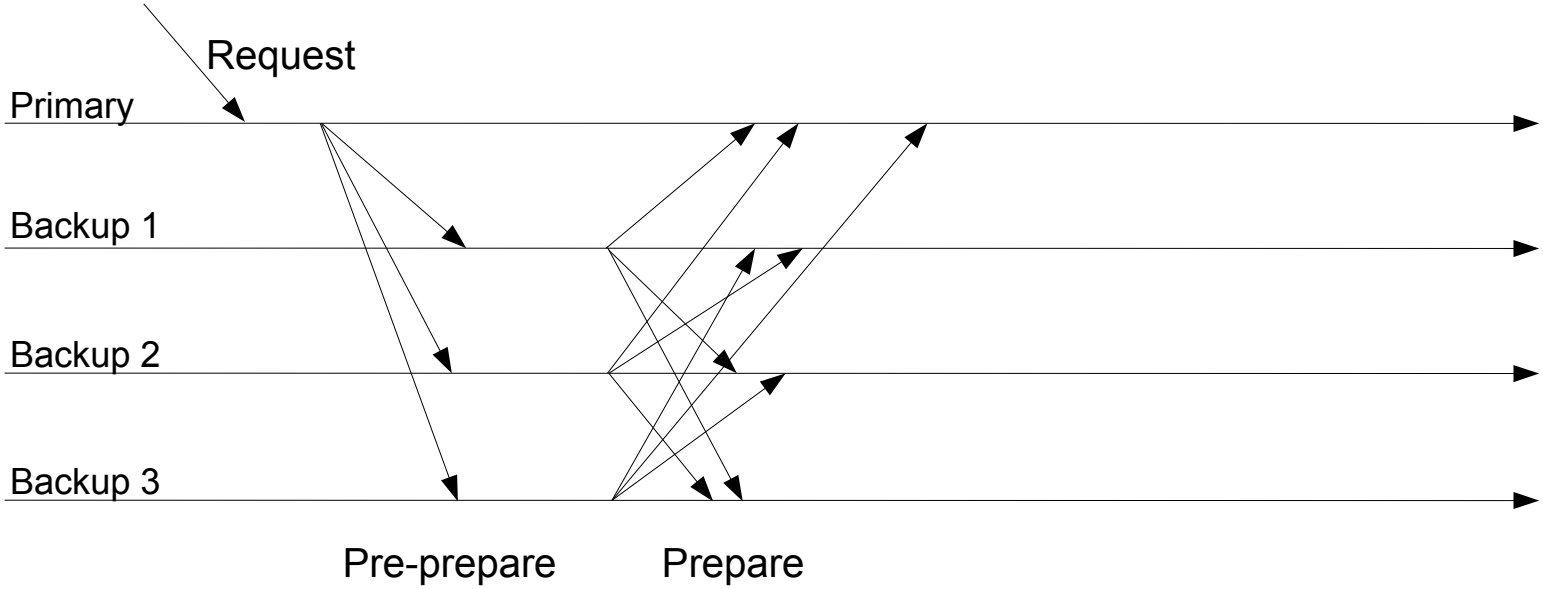
Prepare phase



$\langle \text{PREPARE}, v, n, d(m) \rangle_{\sigma(i)}$

- Accepted by correct replica j if:
  - the PREPARE message is well formed
  - current view of j is v
  - n is between two water-marks L and H

Prepare phase



$\langle \text{PREPARE}, v, n, d(m) \rangle_{\sigma(i)}$

- Replicas that send PREPARE accept the sequence number n for m in view v
- Each accepted PREPARE message is stored in the accepting replica's message log

Prepare certificate (P-certificate)

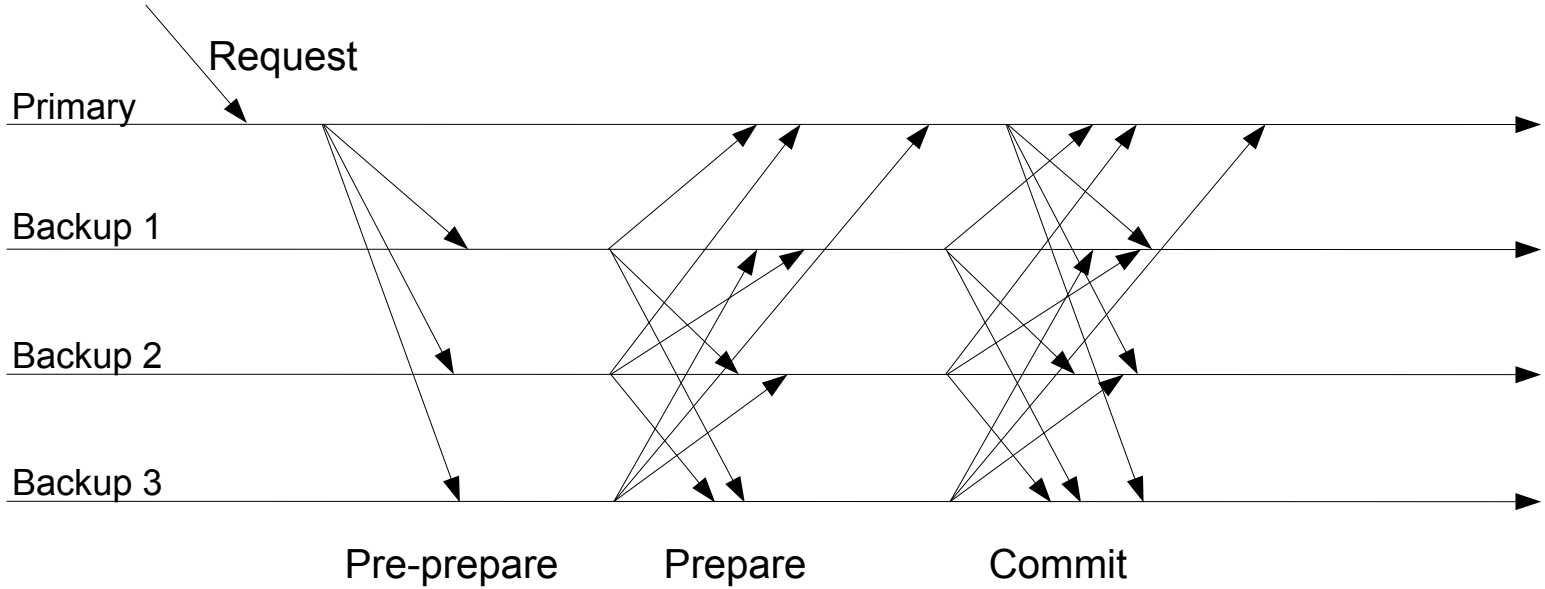
- Replica i produces a prepare certificate **prepared**(m, v, n, i) iff its log holds:
  - The request m
  - A PRE-PREPARE for m in view v with sequence number n
  - Log contains $2f$ PREPARE messages from different backups that match the PRE-PREPARE
- **prepared**(m, v, n, i) means that a quorum of $(2f + 1)$ replicas agrees with assigning sequence number n to m in view v

Theorem

There are no two non-faulty replicas i, j such that **prepared**(m, v, n, i) *and* **prepared**(m', v, n, j)*, with* $m \neq m'$

Proof?

Commit phase



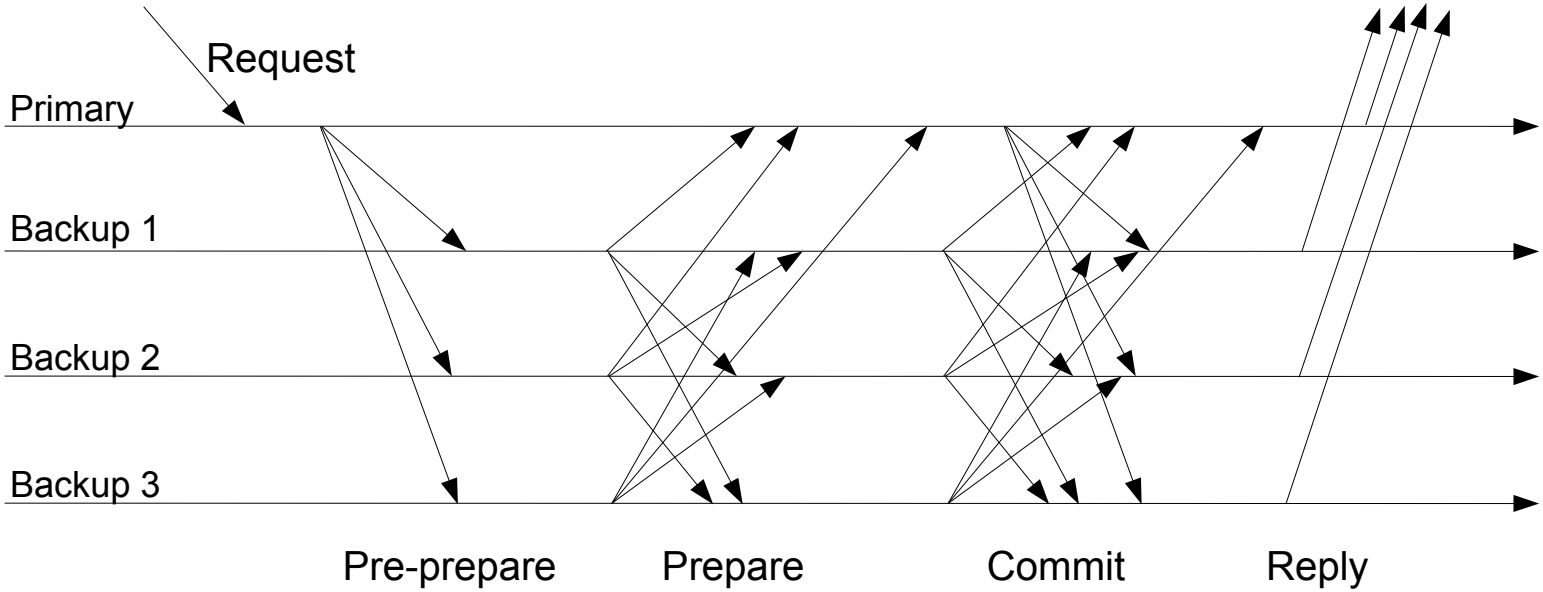
$\langle \text{COMMIT}, v, n, d, i \rangle_{\sigma(i)}$

- After having collected a P-certificate **prepared**(m, v, n, i), replica i sends a COMMIT message
- Accepted if:
  - The COMMIT 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

Commit phase



$\langle \text{COMMIT}, v, t, c, i, r \rangle_{\sigma(i)}$

- r is the reply
- Client waits for $f + 1$ replies with the same t, r
- If the client does not receive replies soon enough, it broadcast the request to all replicas

View change

- 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 the new primary collects a new-view certificate V:
  - a set containing $2f+1$ VIEW-CHANGE messages
  - indicating that $2f+1$ distinct replicas (including itself) support the change of leadership

View change

The "primary elect" p' (replica $v+1 \bmod N$):

- extracts from the new-view certificate V the highest sequence number h of any message for which V contains a P-certificate
- creates a new PRE-PREPARE message for any client message m with sequence number $n \leq h$ and add it to the set O
  - if there is a P-certificate for n, m in V

$$O \leftarrow O \cup \langle \text{PRE-PREPARE}, v+1, n, d_m \rangle_{\sigma(p')}$$

  - Otherwise

$$O \leftarrow O \cup \langle \text{PRE-PREPARE}, v+1, n, d_{null} \rangle_{\sigma(p')}$$

- p' multicasts $\langle \text{NEW-VIEW}, v+1, V, O \rangle_{\sigma(p')}$

View change

- Backup accepts a $\langle \text{NEW-VIEW}, v+1, V, O \rangle_{\sigma(p')}$ message for $v+1$ if
  - it is signed properly by p'
  - V contains valid VIEW-CHANGE messages for $v+1$
  - the correctness of O can be locally verified (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 PREPAREs to the 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 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
- A set S containing $2f + 1$ equivalent CHECKPOINT messages from distinct processes 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 sequence number of the 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 sequence number of the last stable checkpoint
  - V, O: contains only requests with sequence number larger than n

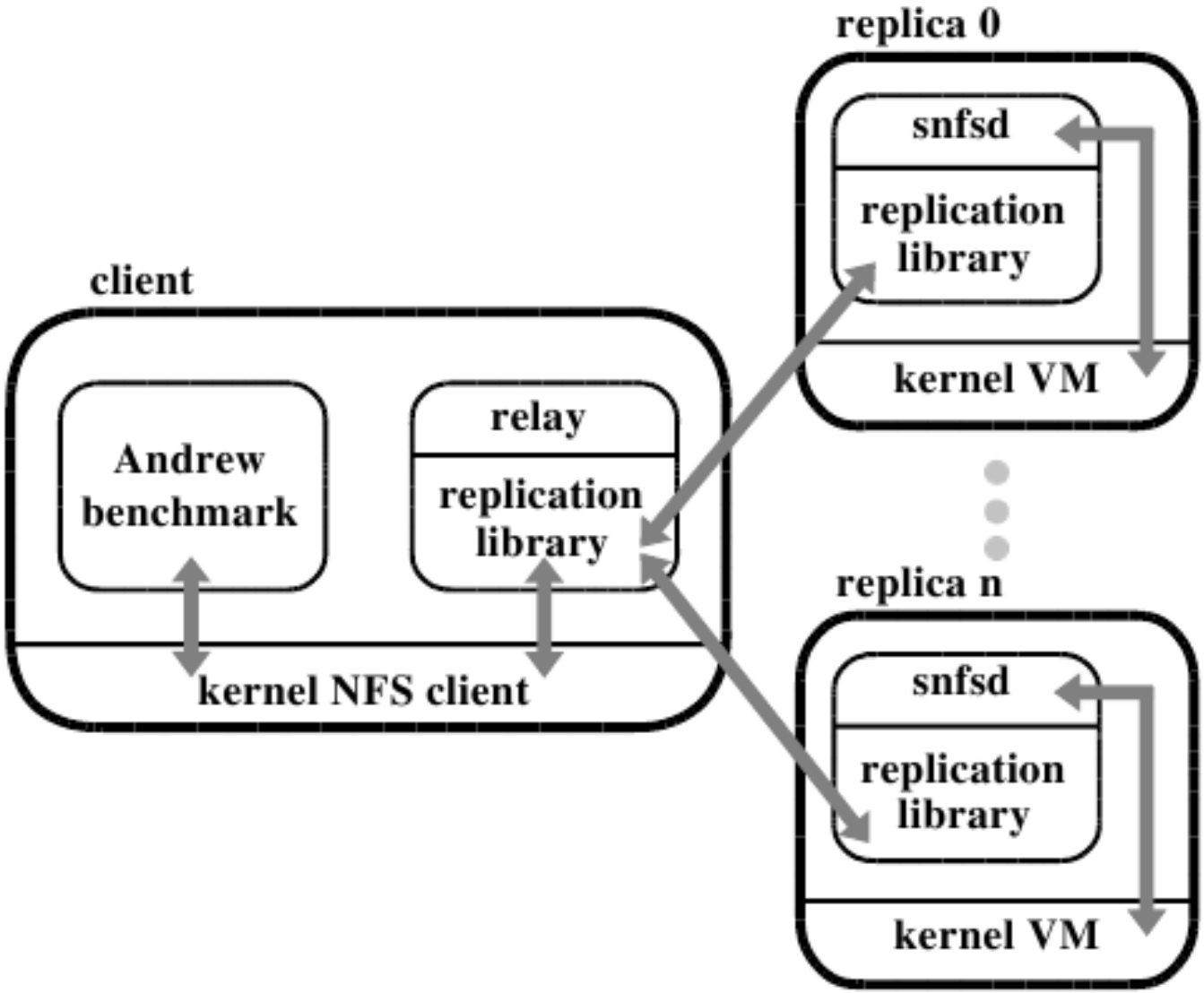
Optimizations

- 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 phase (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

Optimizations: 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: 10μs

Application: Byzantine NFS server



Application: Byzantine NFS server

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.

Contents

1 Introduction
2 Byzantine generals
3 Practical Byzantine fault tolerance
4 Bibliography

Reading material

- M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *Proc. of the 3rd Symposium on Operating systems design and implementation*, OSDI'99, pages 173–186, New Orleans, Louisiana, USA, 1999. USENIX Association. `http://www.disi.unitn.it/~montreso/ds/papers/PbftOsdi.pdf`