Microsoft®
Research

Each year Microsoft Research hosts hundreds of influential speakers from around the world including leading scientists, renowned experts in technology, book authors, and leading academics, and makes videos of these lectures freely available.

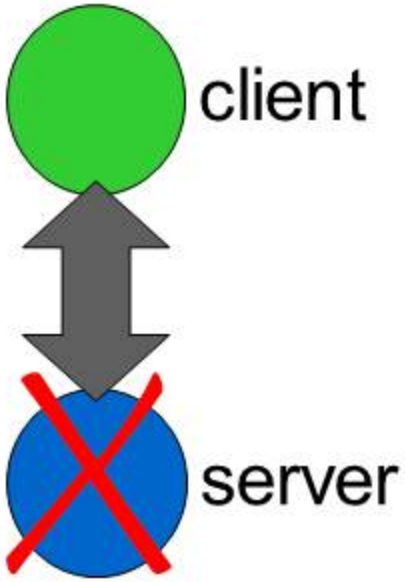
Practical Byzantine Fault Tolerance

Miguel Castro

Microsoft Research Cambridge

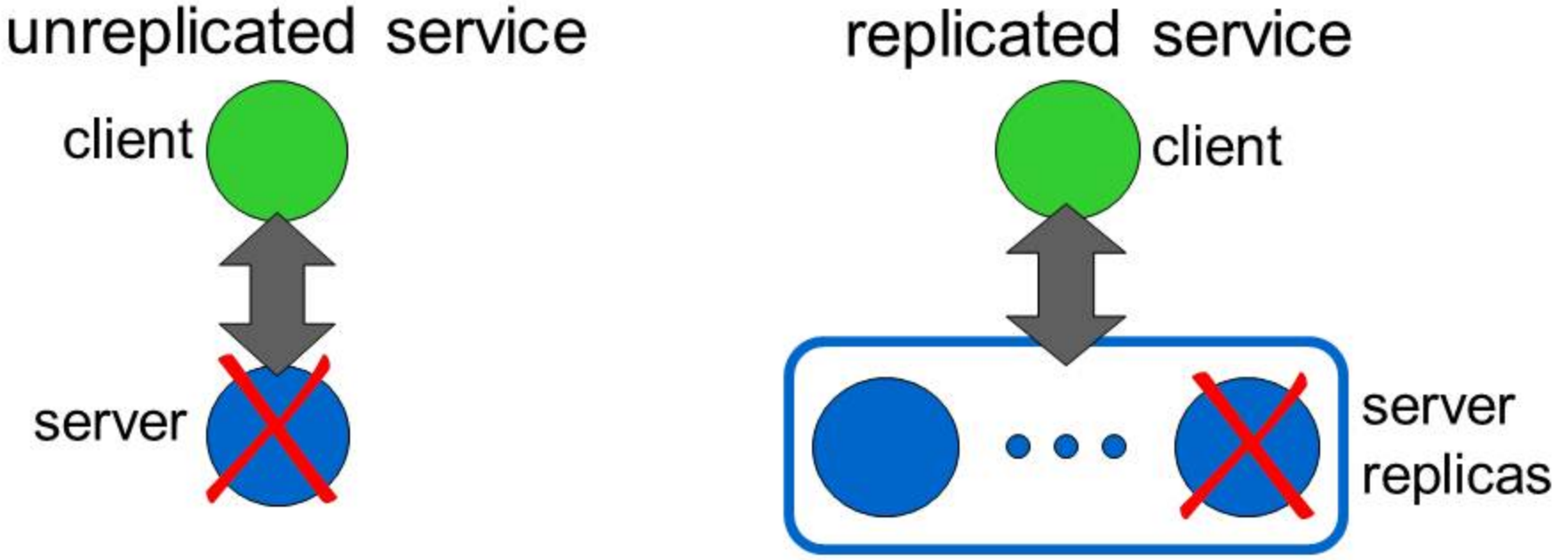
Problem

- Computer systems provide crucial services
- **Computer systems fail**
  - **software errors**
  - **malicious attacks**



Need highly-available services

Replication can help



Replication algorithm:

- masks a fraction of faulty replicas
- high availability if replicas fail “independently”
- software replication allows distributed replicas

Assumptions are a problem

- Replication algorithms make assumptions:
  - behavior of faulty processes
  - synchrony
  - bound on number of faults
- Service fails if assumptions are invalid
  - **attacker will work to invalidate assumptions**

Performance is a problem

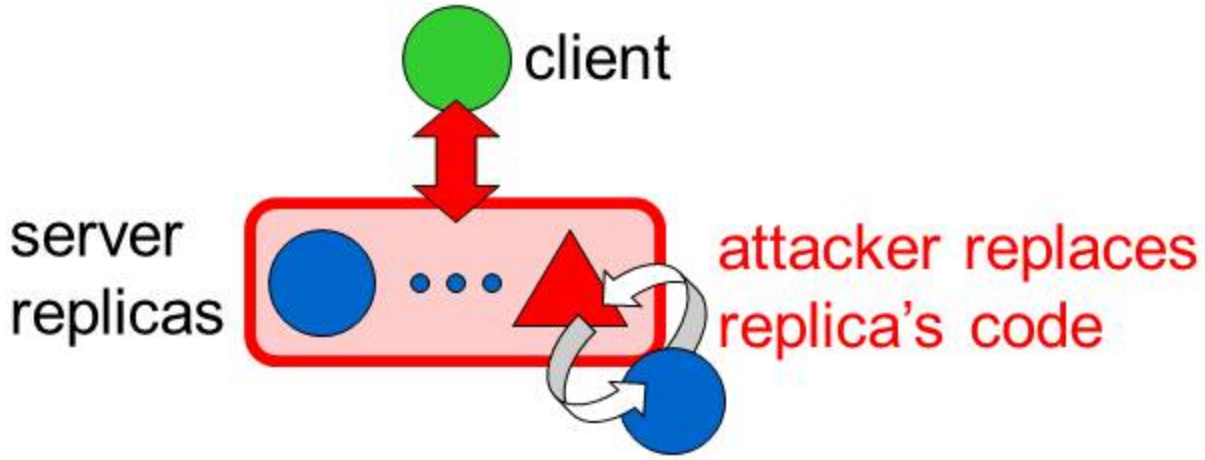
- Replication has performance overhead
  - extra communication and computation
- Practical algorithms require low overhead

Overview

- Problem
- Assumptions
- Algorithm
- Implementation
- Performance
- Other BFT work

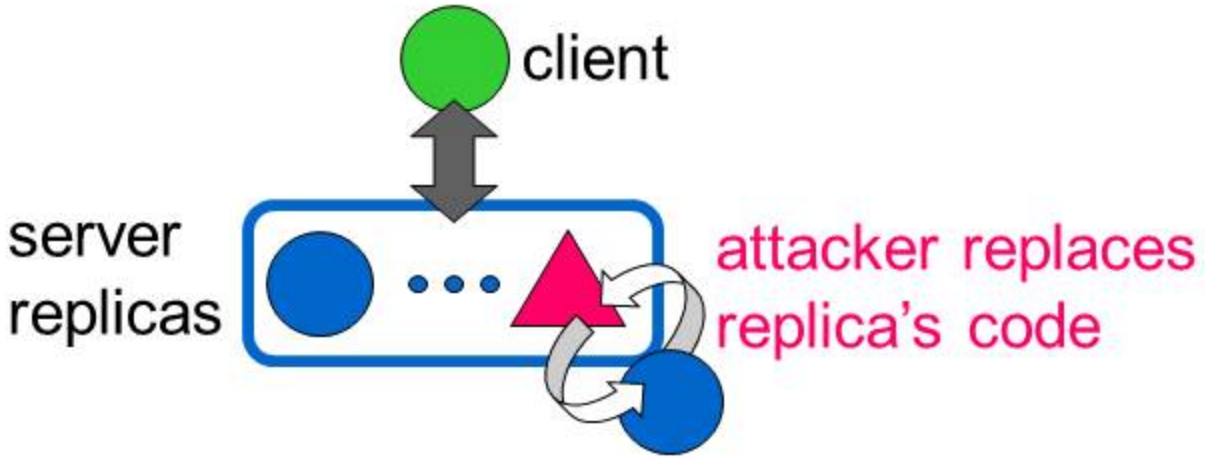
Weak assumption: benign faults

- Traditional replication assumes:
  - replicas fail by stopping or omitting steps
- Invalid with software bugs, malicious attacks:
  - compromised replica may behave arbitrarily
  - single fault may compromise service



Solution: tolerating Byzantine faults

- **Byzantine fault tolerance:**
  - **no assumptions about faulty behavior**
- Tolerates successful attacks
  - service available when hacker controls replicas



Weak assumption: synchrony

- **Synchrony ≡ known bounds** on:
  - delays between steps
  - message delays
- Invalid with denial-of-service attacks:
  - bad replies due to increased delays

Solution: Asynchrony

- **No bounds on delays**
- Problem: replication is impossible

Solution in PBFT:

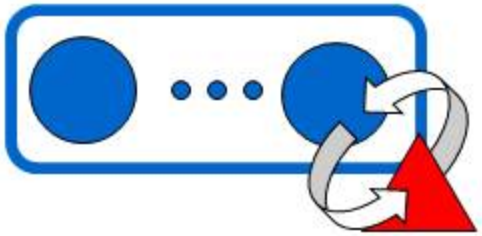
- provide safety without synchrony
  - guarantees no bad replies
- assume eventual time bounds for liveness
  - may not reply with active denial-of-service attack
  - will reply when denial-of-service attack ends

Weak assumption: bound on faults

- Assume at most ***f*** faulty **over system lifetime**
  - more than *f* faults likely given enough time
  - hard to detect Byzantine faults until it is too late
- Faults may not be independent
  - identical replicas have identical bugs
  - replicas may be administered by the same person

Two partial solutions

- Abstraction for code reuse and diversity
  - replicas can be non-deterministic
  - they can run different off-the-shelf implementations
  - use abstraction to make them look the same
- Proactive recovery
  - frequent recovery even if no fault suspected
  - **all replicas can fail** if less than ***f*** fail in window

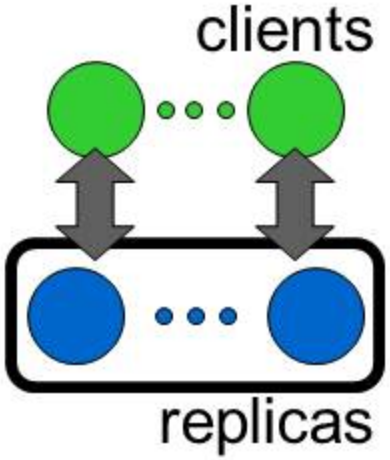


recovery makes faulty replicas behave correctly again

Overview

- Problem
- Assumptions
- Algorithm
- Implementation
- Performance
- Other BFT work

Algorithm properties

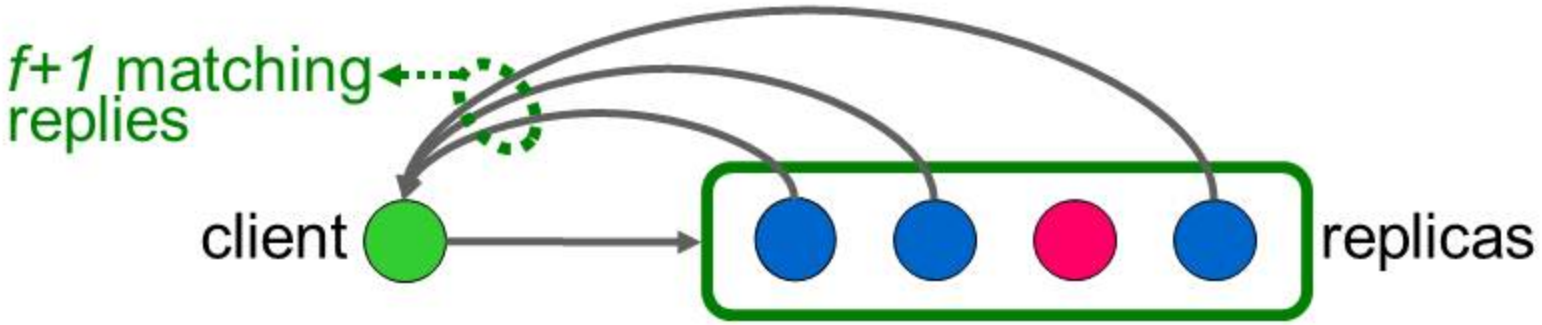


- Arbitrary replicated service
  - complex operations
  - mutable shared state
- Properties (safety and liveness):
  - system behaves as correct centralized service
  - clients eventually receive replies to requests
- Assumptions:
  - *3f+1* replicas to tolerate *f* Byzantine faults (optimal)
  - strong cryptography
  - **only for liveness:** eventual time bounds

Algorithm overview

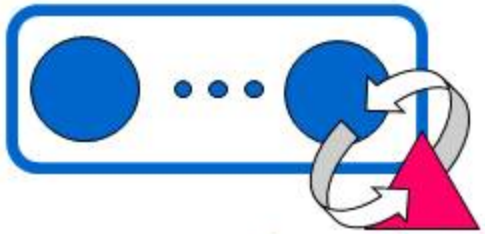
State machine replication:

- deterministic replicas start in same state
- replicas execute same requests in same order
- correct replicas produce identical replies



Hard: ensure requests execute in same order

Recovery



recovery makes faulty replica behave correctly

- Proactive recovery
  - periodic recovery even if no fault suspected
- Frequent recoveries
- **Stronger safety and liveness:**
  - **all replicas can fail if at most *f* fail in window**

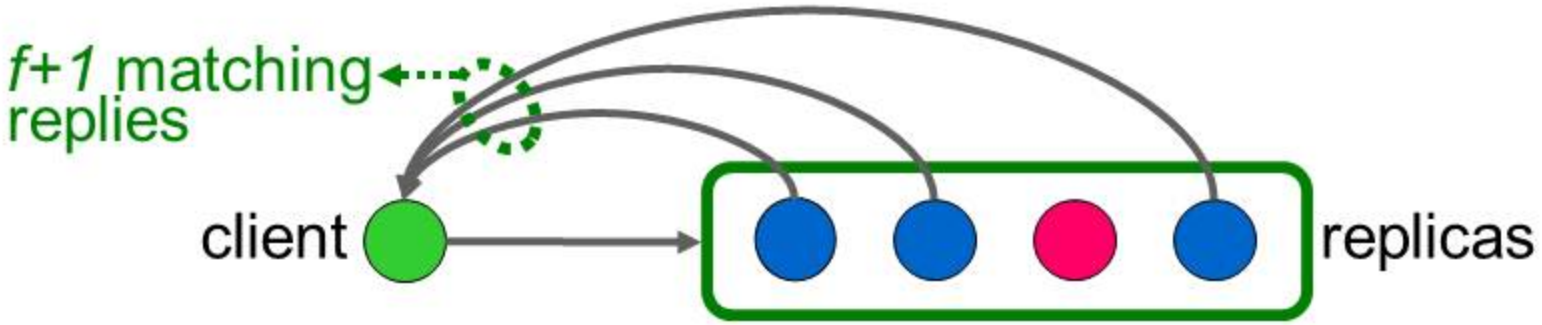
Challenges:

- prevent impersonation by attacker
- ensure it is safe to recover a non-faulty replica
- achieve low latency and low overhead

Algorithm overview

State machine replication:

- deterministic replicas start in same state
- replicas execute same requests in same order
- correct replicas produce identical replies

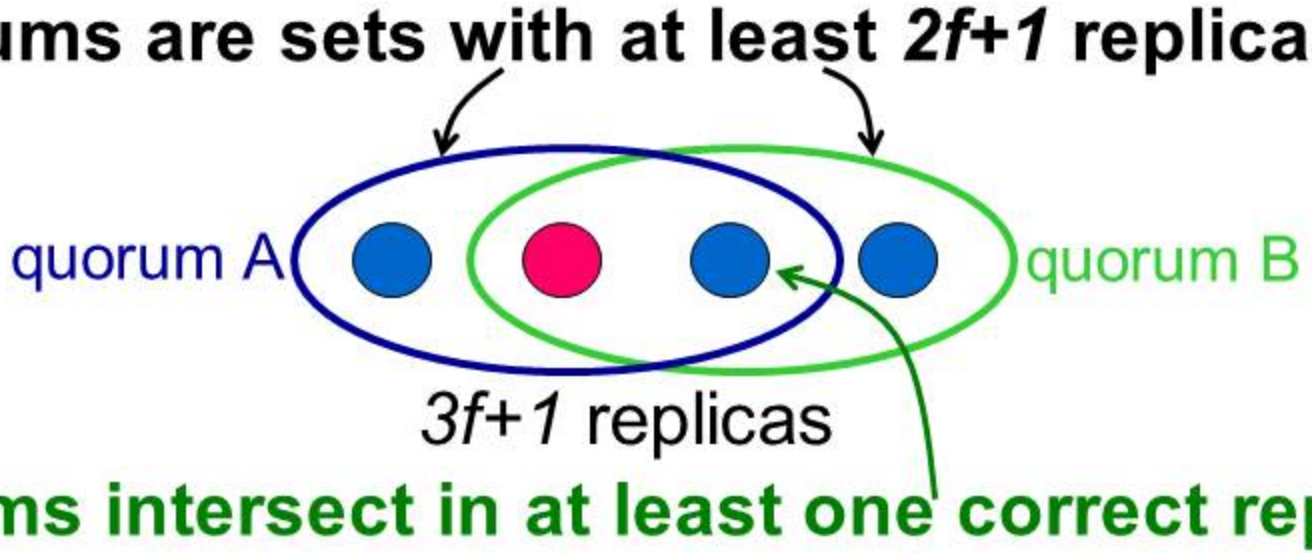


Hard: ensure requests execute in same order

Ordering requests

- Views
  - a view is an integer
  - replicas go through sequence of increasing views
  - view designates primary replica i (i = v mod N)
  - other replicas are backups
- Primary picks request ordering
- Backups ensure primary behaves correctly
  - certify correct ordering
  - trigger view changes to replace faulty primary

Quorums and Certificates



- **Certificate** ≡ set with messages from a quorum
- Algorithm steps are justified by certificates

Algorithm

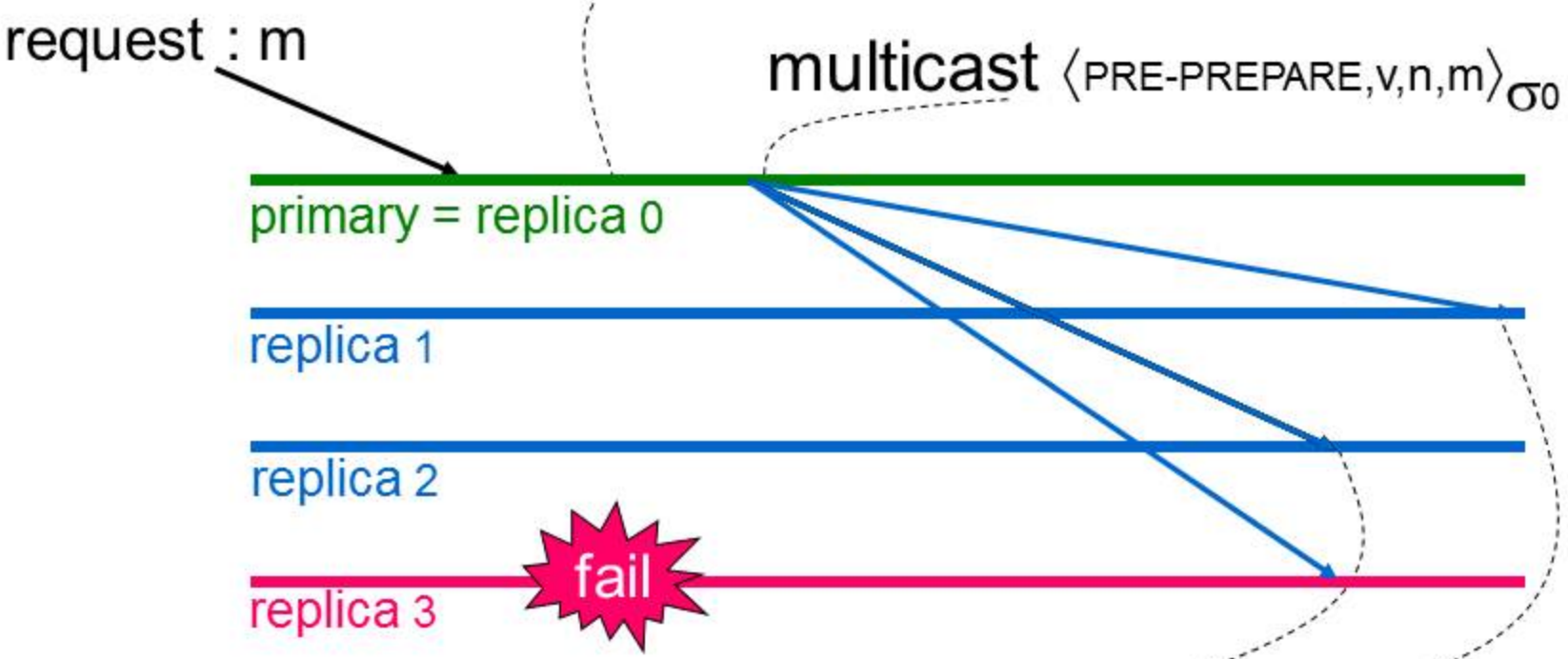
- Normal case operation
- View changes
- View changes without signatures
- Garbage collection
- Optimizations

Normal case operation

- Three phase algorithm:
  - *pre-prepare* picks order of requests
  - *prepare* ensures order within views
  - *commit* ensures order across views
- Replicas remember messages in log
- Messages are signed
  - $\langle \bullet \rangle_{\sigma k}$ denotes a message signed by k

Pre-prepare phase

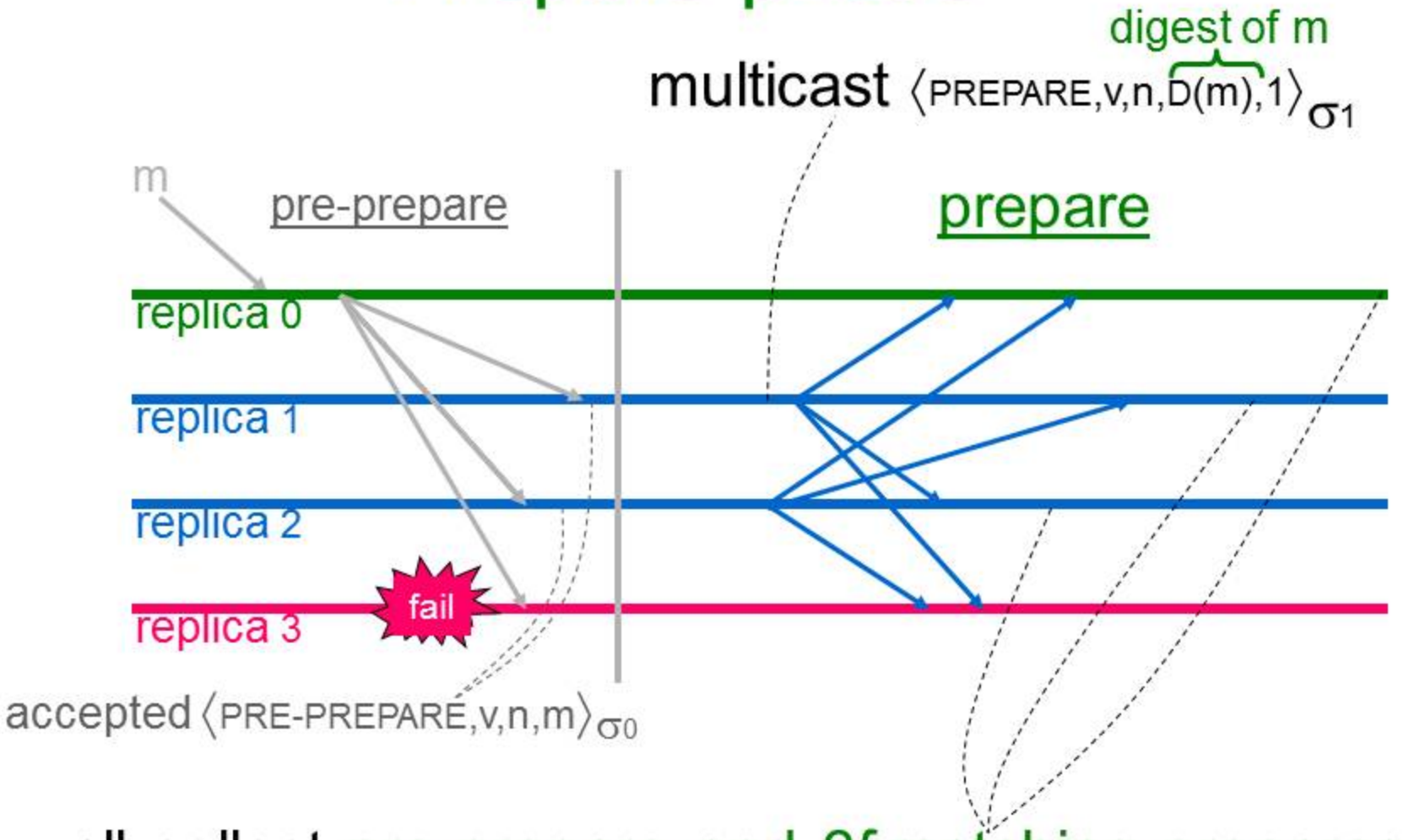
assign sequence number n to request m in view v



backups accept pre-prepare if:

- in view v
- never accepted pre-prepare for v,n with different request

Prepare phase



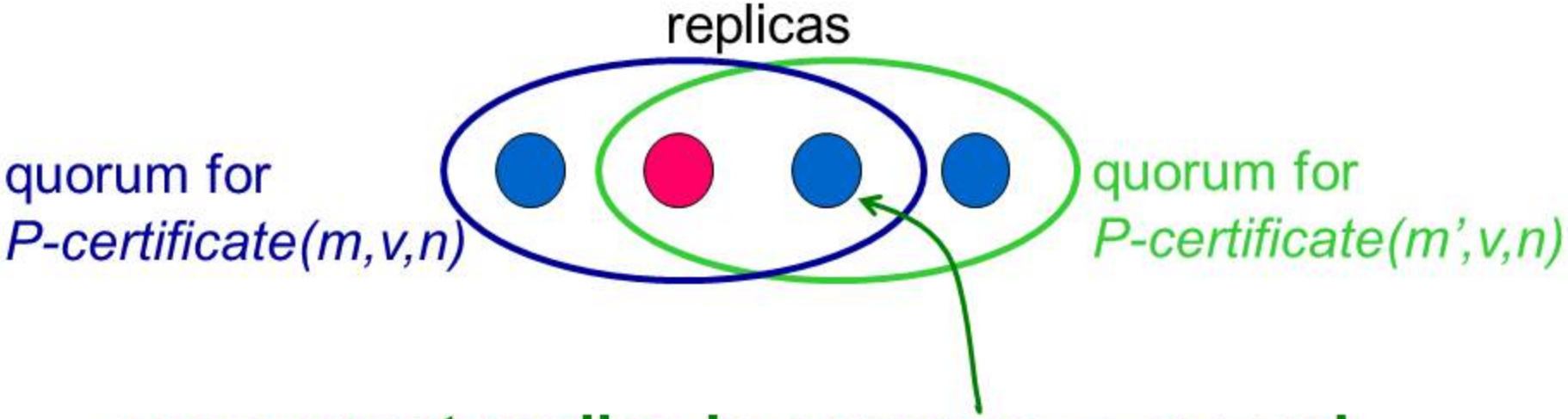
all collect pre-prepare and *2f* matching prepares

P-certificate(m,v,n)

Order within a view

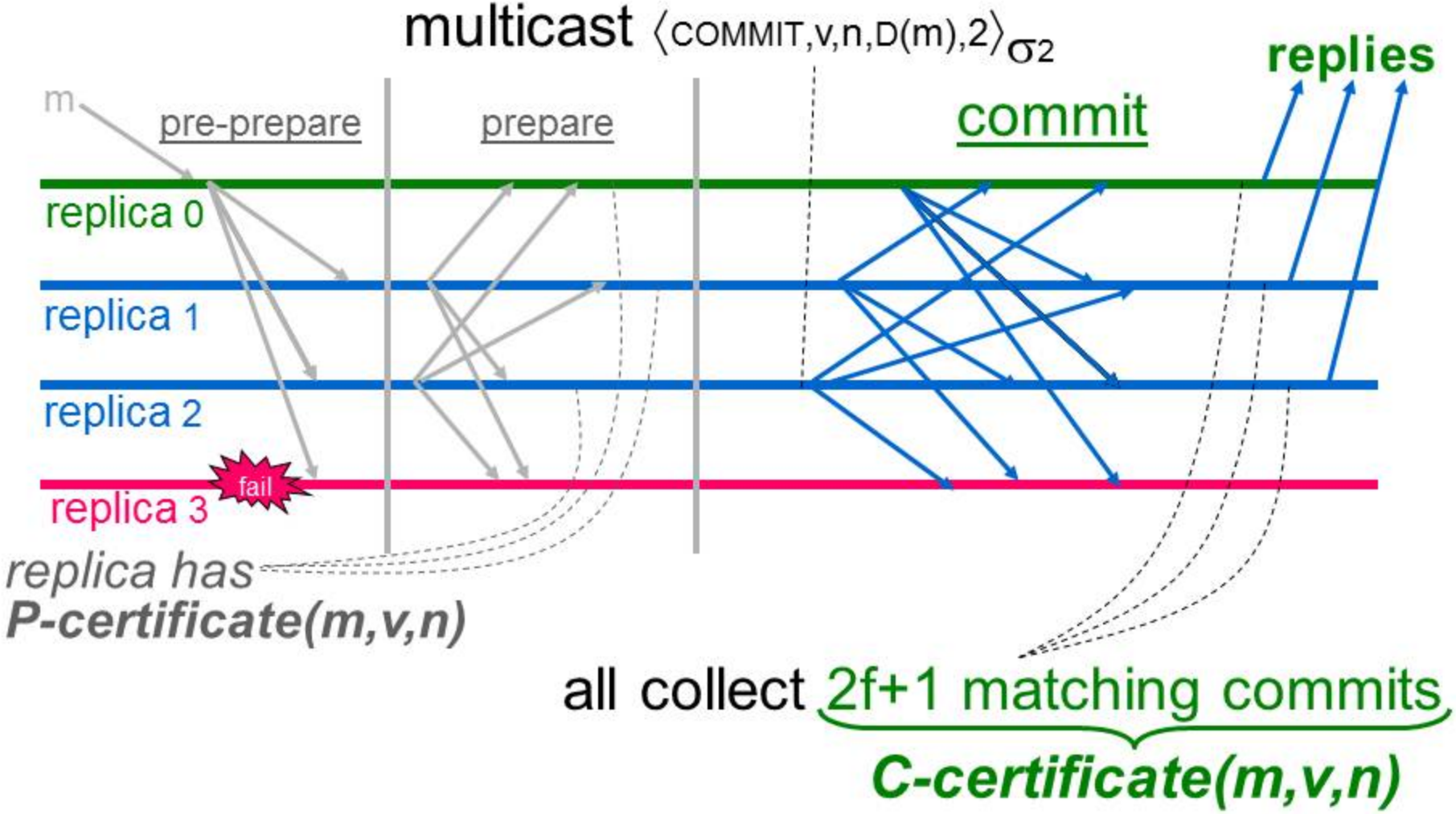
No *P-certificates* with the same view and sequence number and different requests

If it were false:



one correct replica in common ⇒ m = m'

Commit phase



Request execution

- A replica executes request m after:
  - having *C-certificate(m,v,n)*
  - executing requests with lower sequence numbers
- Each replica sends a reply to the client
- Client collects replies from replicas
  - returns result that appears in at least f+1 replies

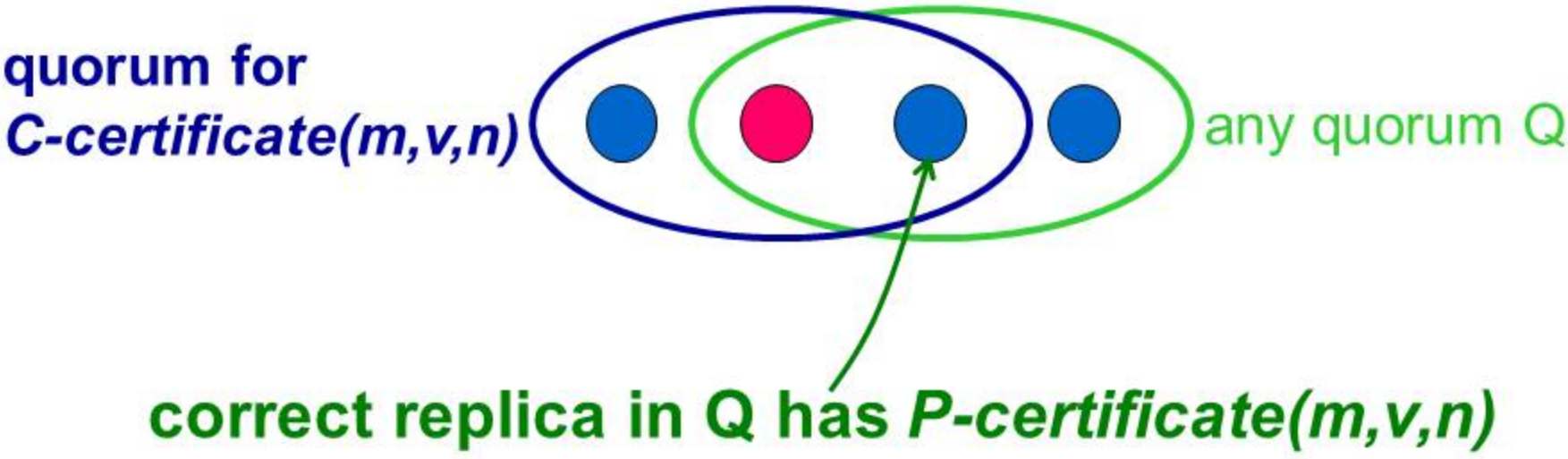
View changes

- Provide liveness when primary fails:
  - timeouts trigger view changes
  - select new primary (≡ view number mod *3f+1)*
- But also need to:
  - preserve safety
  - ensure replicas are in the same view long enough
  - prevent denial-of-service attacks

View change safety

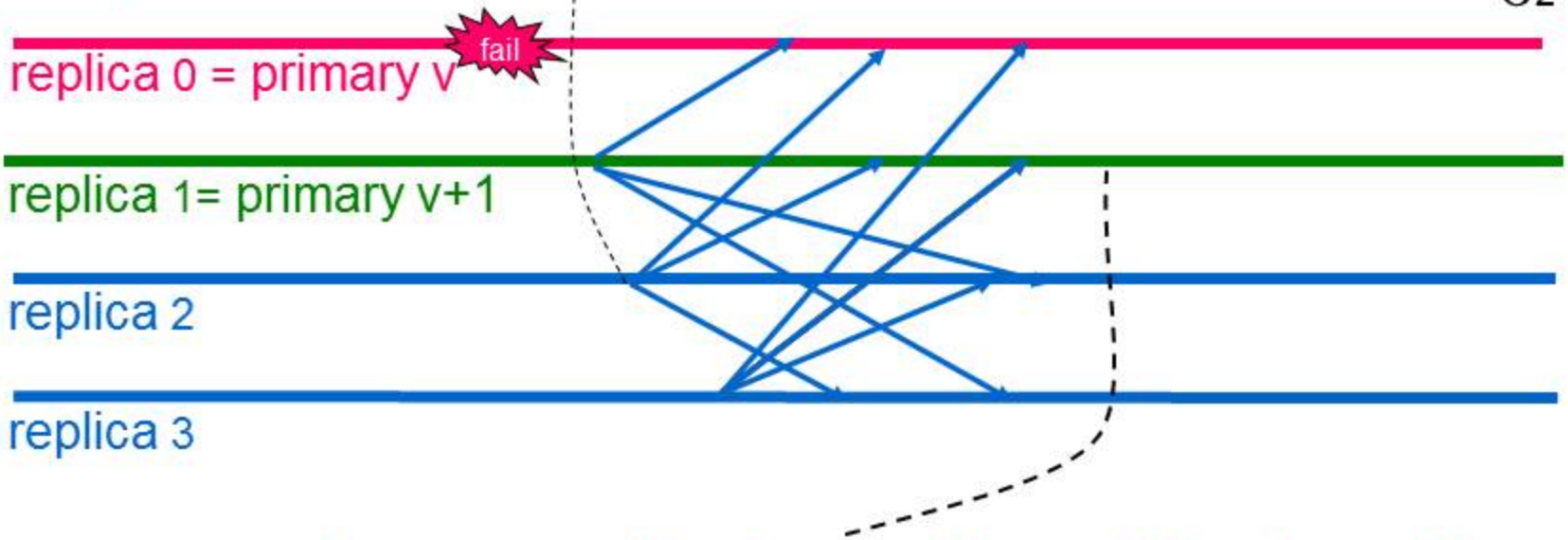
Goal: No *C-certificates* with the same sequence number and different requests

- Intuition: if replica has ***C-certificate(m,v,n)*** then



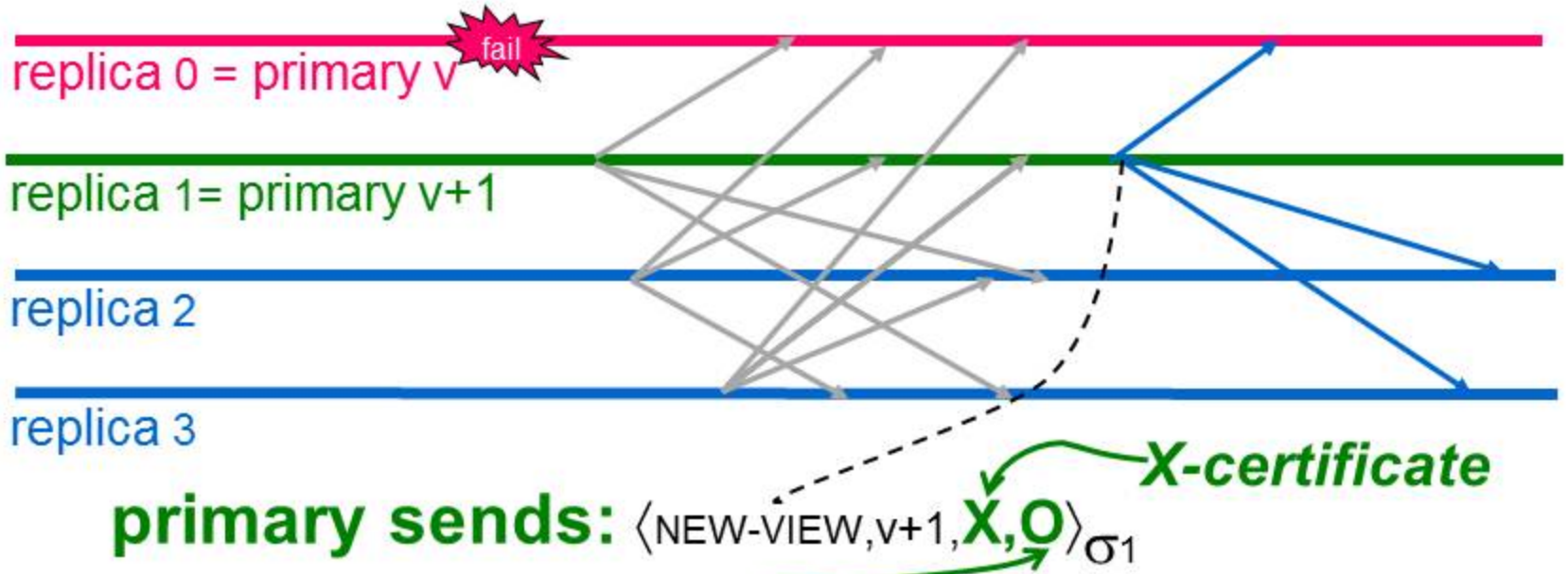
View change protocol

replicas send *P-certificates*: $\langle \text{VIEW-CHANGE}, v+1, \mathbf{P}, 2 \rangle_{\sigma 2}$



primary collects an *X-certificate with* 2f+1 view change messages

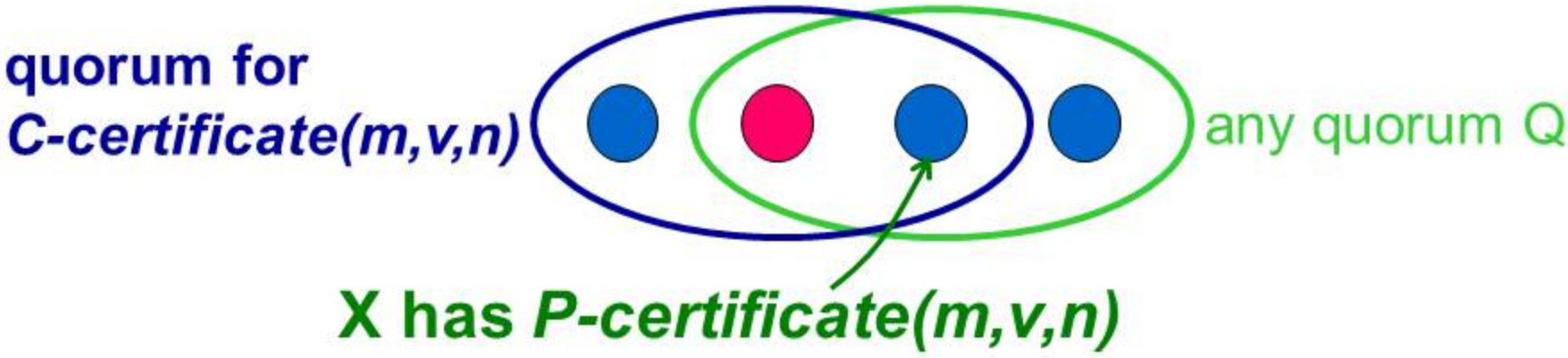
View change protocol



New view message

- **O** has ⟨PRE-PREPARE,v+1,n,m⟩ s.t.
  - **P-certificate(m,v',n)** in **X**
  - there is no **P-certificate(*,v'',n)** with v'' > v'
- or ⟨PRE-PREPARE,v+1,n,**nop**⟩
  - if there is no **P-certificate(*,*,n)** in **X**
- backups:
  - check **O** and **X** in the new view message
  - broadcast prepares for pre-prepares in **O**

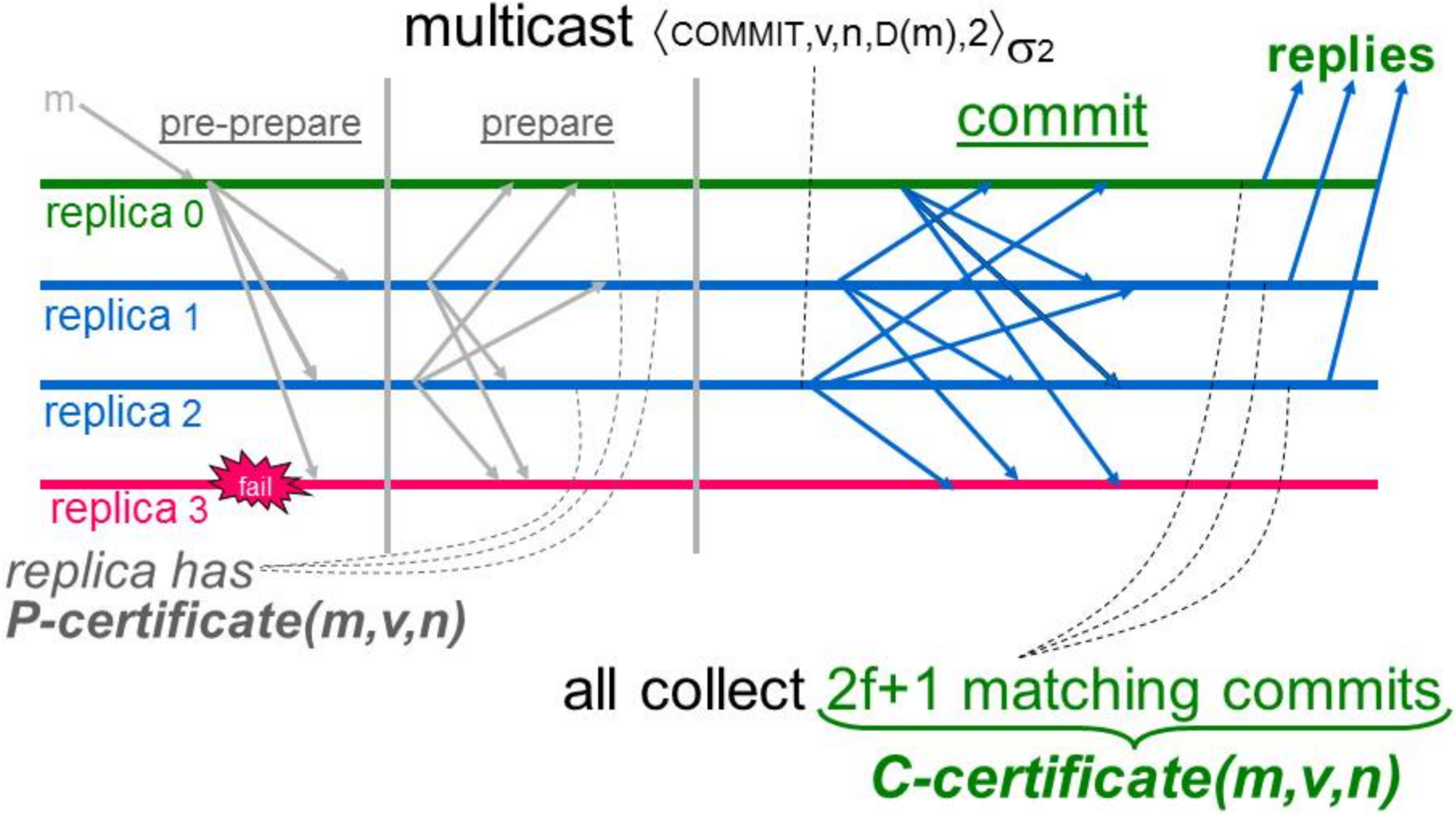
View change safety



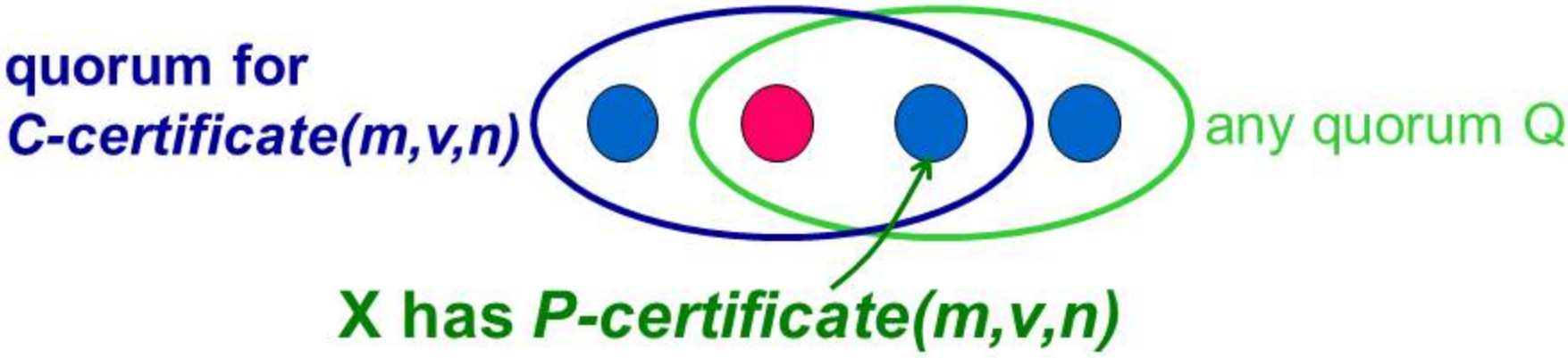
- therefore ⟨**PRE-PREPARE,v+1,n,m**⟩ ∈ **O**
- backups will not accept another pre-prepare for n
- therefore by induction for all v' > v,
  - new-view messages include ⟨PRE-PREPARE,v',n,m⟩

So if m executes at n, no m' ≠ m is assigned n

Commit phase



View change safety



- therefore ⟨**PRE-PREPARE,v+1,n,m**⟩ ∈ **O**
- backups will not accept another pre-prepare for n
- therefore by induction for all v' > v,
  - new-view messages include ⟨PRE-PREPARE,v',n,m⟩

So if m executes at n, no m' ≠ m is assigned n

New view message

- **O** has ⟨PRE-PREPARE,v+1,n,m⟩ s.t.
  - **P-certificate(m,v',n)** in **X**
  - there is no **P-certificate(*,v'',n)** with v'' > v'
- or ⟨PRE-PREPARE,v+1,n,**nop**⟩
  - if there is no **P-certificate(*,*,n)** in **X**
- backups:
  - check **O** and **X** in the new view message
  - broadcast prepares for pre-prepares in **O**

Algorithm

- Normal case operation
- View changes
- View changes without signatures
- Garbage collection
- Optimizations

Fast authentication

- Use MACs instead of digital signatures
  - $\langle \bullet \rangle_{\alpha k}$ denotes message with vector of MACs appended
- Public-key cryptography only to setup MAC keys
- **MAC 1000x faster than public-key signatures**
- Non-trivial: MACs less powerful than signatures
  - MAC proves to receiver sender sent a message
  - receiver cannot prove authenticity to others

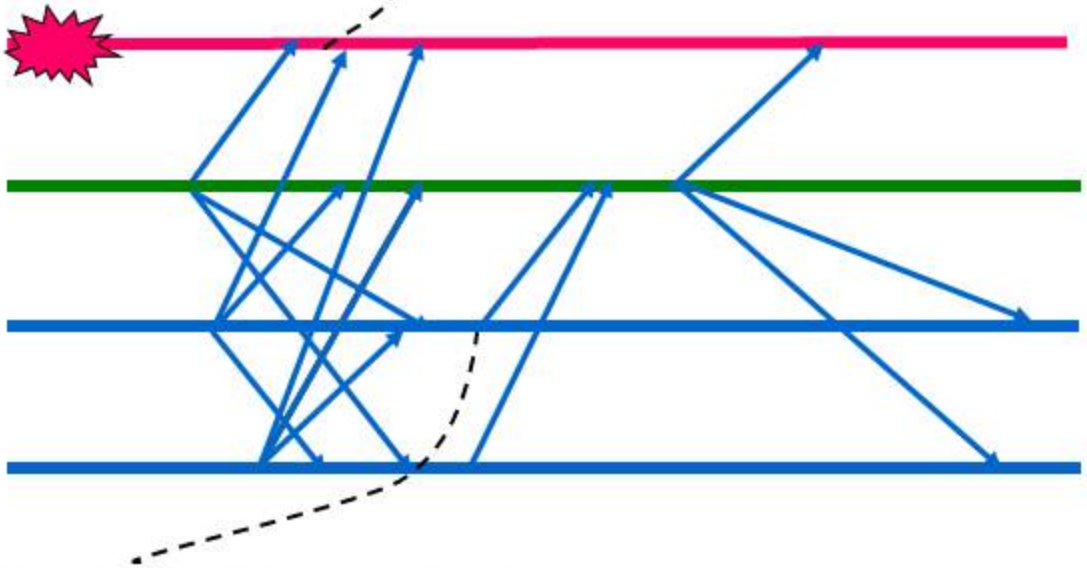
View changes without signatures

- View change protocol relied on certificates:
  - *P-certificates and X-certificates*
- Certificates are useless without signatures
  - individual replica can verify message authenticity
  - cannot prove set of messages is a certificate

New view change protocol

⟨VIEW-CHANGE,v+1,**P**,**Q**,2⟩$_{\alpha 2}$

- **P** has ⟨n,d,v⟩ for every P-certificate collected by the sender
- **Q** has ⟨n,d,v⟩ for every prepare or pre-prepare sent



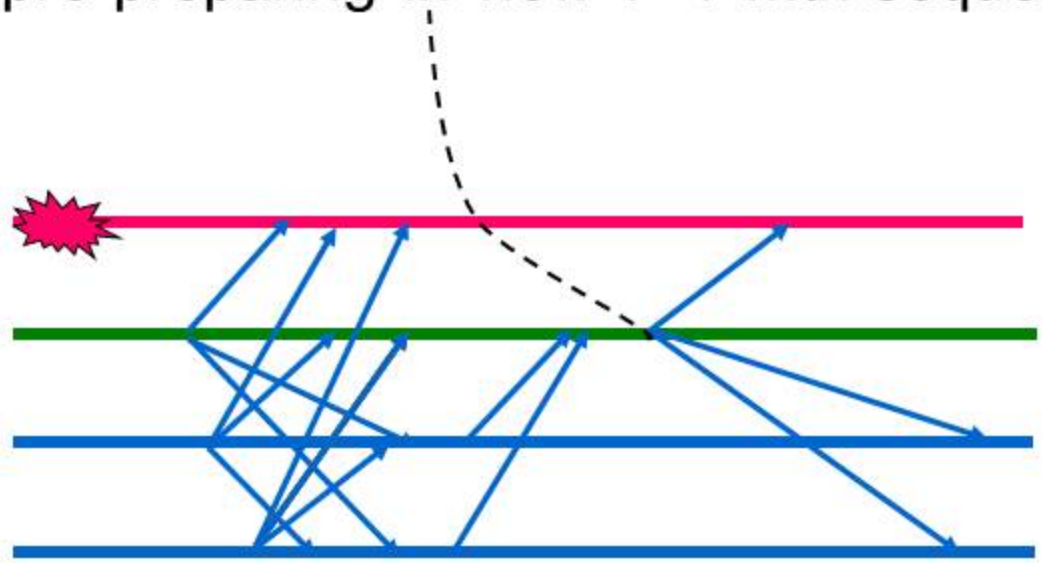
⟨VIEW-CHANGE-ACK,v+1,2,3,d⟩

- 2 has received a view change message from 3 with digest d
- replace signatures in view-change messages

New view change protocol

$\langle$NEW-VIEW-CHANGE,v+1,**V**,**O**,1$\rangle_{\alpha 1}$

- **V** has at least 2f+1 pairs $\langle$i,d$\rangle$ where d is the digest of a view change message and i the replica that sent it
- **O** has pairs <n,D(m)> where m is a request that the new primary is pre-preparing in view v+1 with sequence number n



What is different?

- No P-certificates in view-change messages
  - no point because messages are not signed
- Replicas *claim* they collected P-certificates
  - ‹n,D(m),v› ∈ **P** claims ***P-certificate(m,v,n)***
  - *but faulty replicas can lie*
- The additional information in **Q** constrains lies
  - ‹n,D(m),v› ∈ **Q** claims replica sent matching pre-prepare or prepare
  - claims in **P**s must be ***supported*** by claims in **Q**s

Supporting claims

- ‹n,D(m),v› ∈ **P_i** is **supported** iff
  - f+1 view change messages have ‹n,D(m),v› ∈ **Q_j**
- Correct claims eventually get support
  - P-certificate(m,v,n) has 2f+1 messages
  - f+1 messages come from correct replicas
- Incorrect claims must be consistent with past
  - since at least a correct replica has ‹n,D(m),v› ∈ **Q_j**
  - must be consistent with commits in previous views

Supported lies

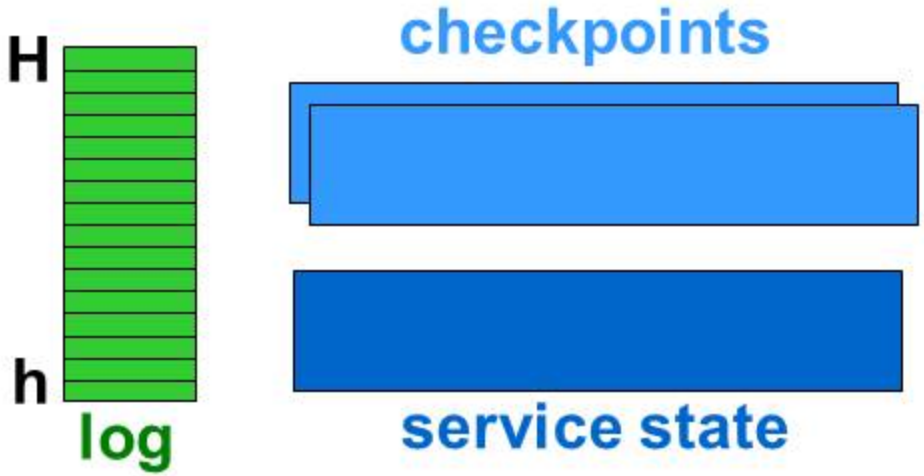
- Incorrect claims may be supported
  - ‹n,D(m),v› $\in$ $\mathbf{P_i}$ ‹n,D(m'),v› $\in$ $\mathbf{P_j}$ may be supported
  - how do we pick the right claim ?
- The solution is to wait for correct replicas
  - usually cannot wait for more than n-f messages
  - but there are 2f+1 correct replicas that do not lie
  - new primary waits for 2f+1 view change messages without conflicting claims about prepared requests

Algorithm

- Normal case operation
- View changes
- View changes without signatures
- Garbage collection
- Optimizations

Bounding replica state

- Cannot keep messages indefinitely
- Log size is bounded (H = h + L)
- Checkpoints are snapshots of state
- h is sequence number of stable checkpoint
  - checkpoint is stable when reached by a quorum



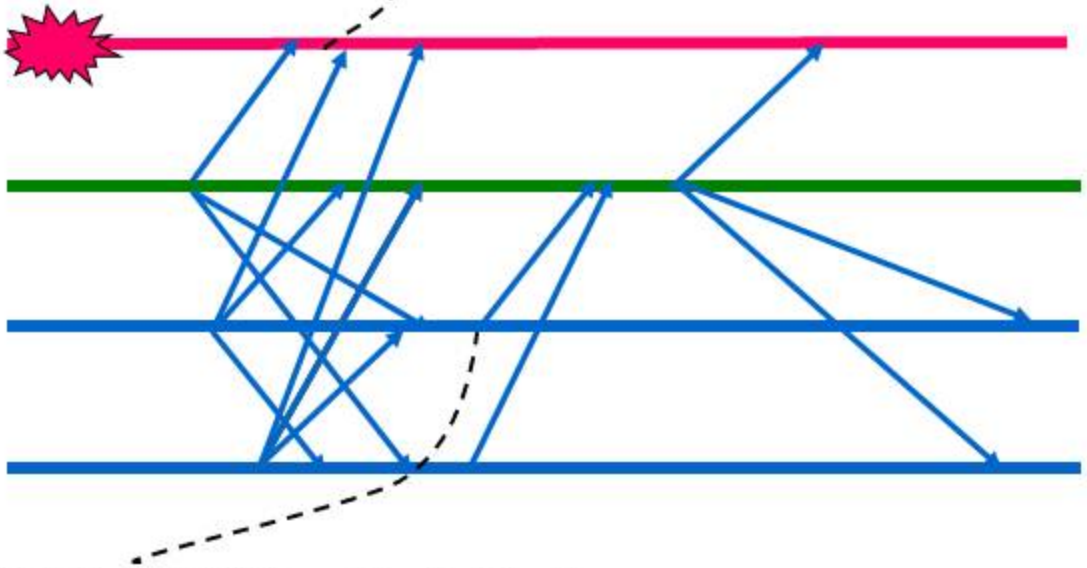
Fast authentication

- Use MACs instead of digital signatures
  - $\langle \bullet \rangle_{\alpha k}$ denotes message with vector of MACs appended
- Public-key cryptography only to setup MAC keys
- **MAC 1000x faster than public-key signatures**
- Non-trivial: MACs less powerful than signatures
  - MAC proves to receiver sender sent a message
  - receiver cannot prove authenticity to others

New view change protocol

⟨VIEW-CHANGE,v+1,**P**,**Q**,2⟩$_{\alpha 2}$

- **P** has ⟨n,d,v⟩ for every P-certificate collected by the sender
- **Q** has ⟨n,d,v⟩ for every prepare or pre-prepare sent



⟨VIEW-CHANGE-ACK,v+1,2,3,d⟩

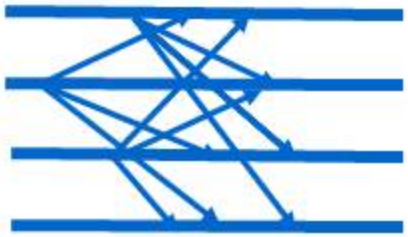
- 2 has received a view change message from 3 with digest d
- replace signatures in view-change messages

Garbage collection

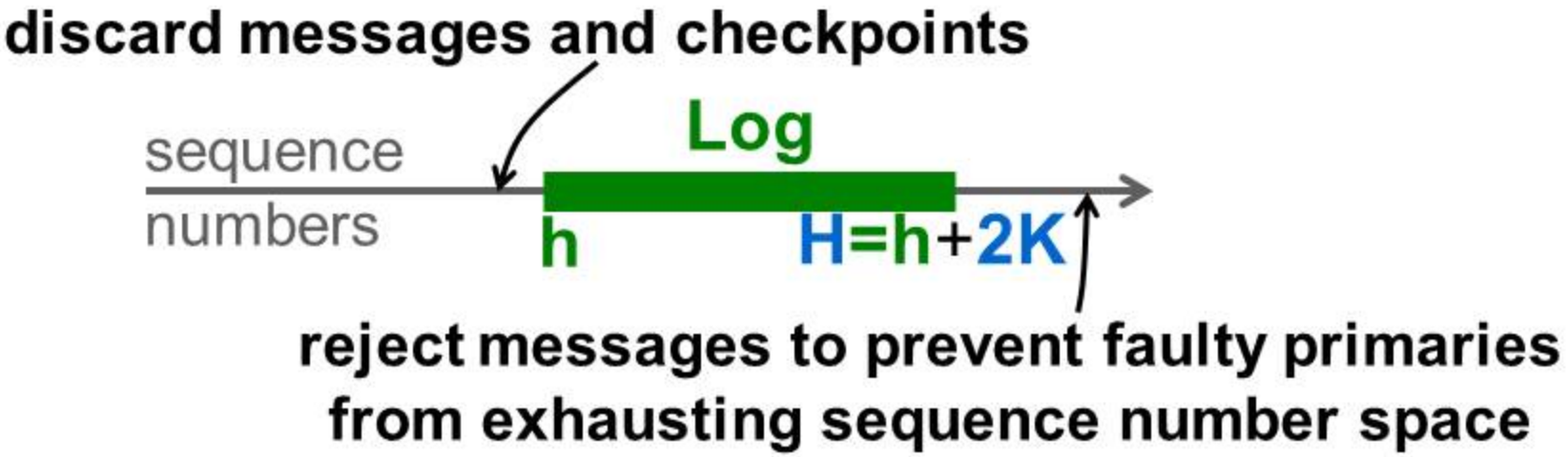
Truncate log with **certificate**:

- periodically checkpoint state (**K**)
- multicast $\langle \text{CHECKPOINT}, h, D(\text{checkpoint}), i \rangle_{\alpha_i}$
- all collect *2f+1* checkpoint messages

  S-certificate(h,checkpoint)
- checkpoint with S-certificate is stable



Garbage collection



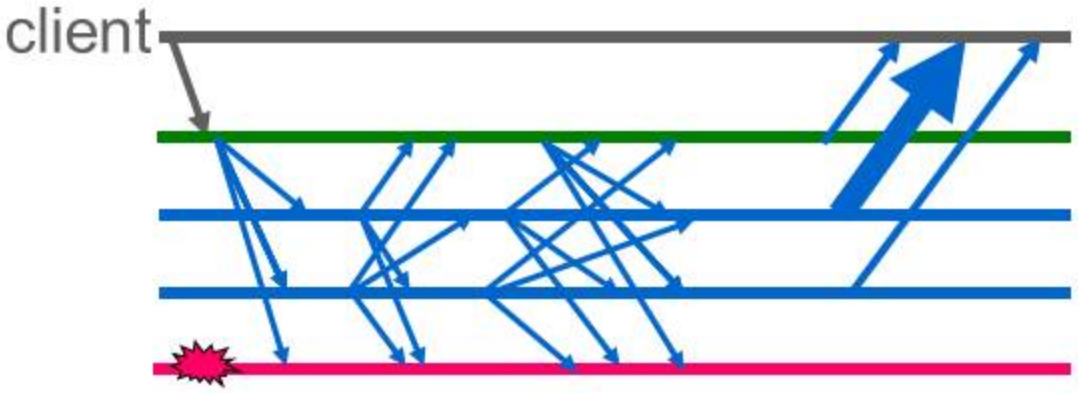
- View-change messages modified to
  - include checkpoint sequence numbers and digests
  - remove information for sequence numbers less than h

Algorithm

- Normal case operation
- View changes
- View changes without signatures
- Garbage collection
- Optimizations

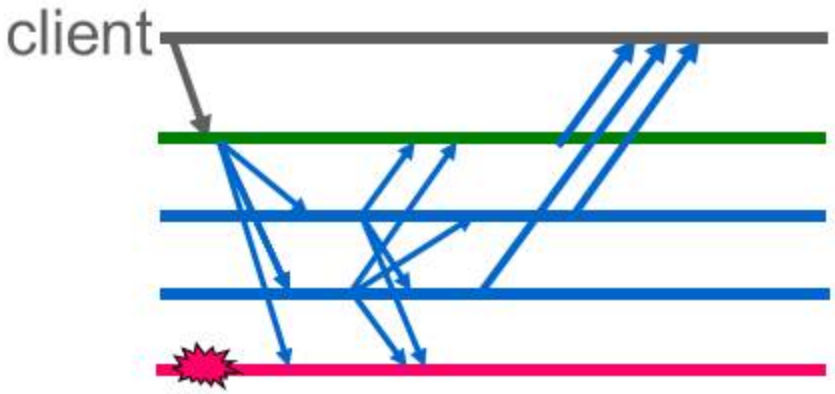
Communication optimizations

- Digest replies:
  - only one replica sends the result in the reply message
  - other replicas send result digests
  - change replica designated to send result on retransmission
  - balances bandwidth utilization across replicas



Communication optimizations

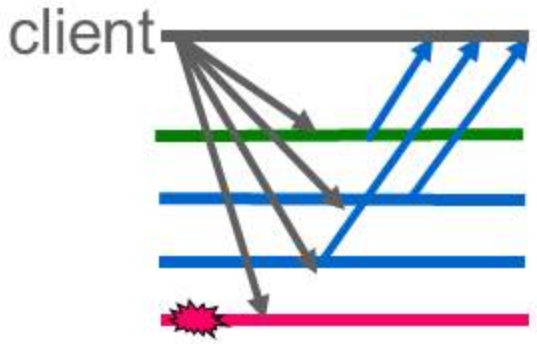
- Optimistic execution:
  - execute prepared requests (after getting P-certificate)
  - rollback execution of uncommitted requests in view changes
  - clients must collect 2f+1 matching replies
  - can piggyback commits in pre-prepares and prepares



Read-write operations execute in two round-trips

Communication optimizations

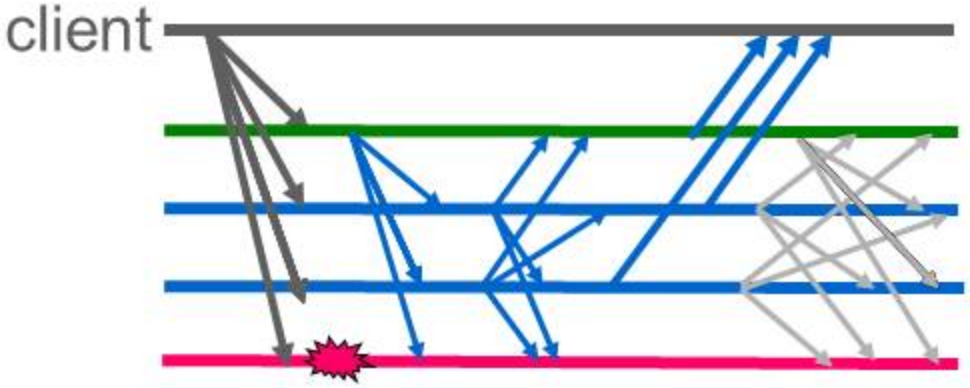
- Read-only optimization:
  - read-only operations executed in current state
  - clients must collect 2f+1 matching replies
  - retransmit request as read-write if this fails



Read-only operations
execute in one round-trip

Communication optimizations

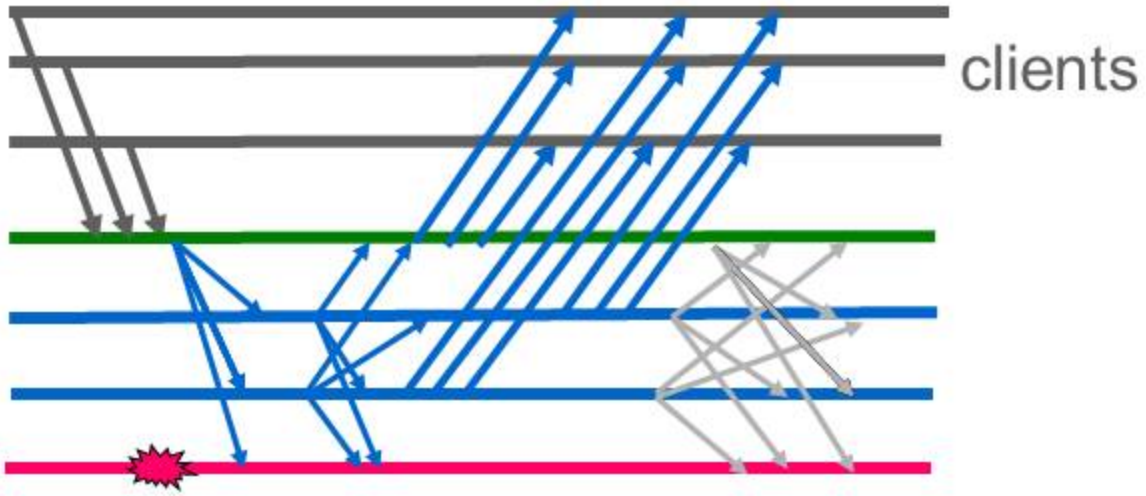
- Separate request transmission:
  - clients multicast large requests to all replicas
  - other protocol messages include only request digests
  - increases parallelism in request processing
  - reduces cost of processing pre-prepare messages



messages include only request digest

Communication optimizations

- Batching:
  - run the algorithm on a batch of requests
  - amortizes cost across the batch
  - use sliding window to minimize impact on latency
  - significantly improves throughput



Reliable communication

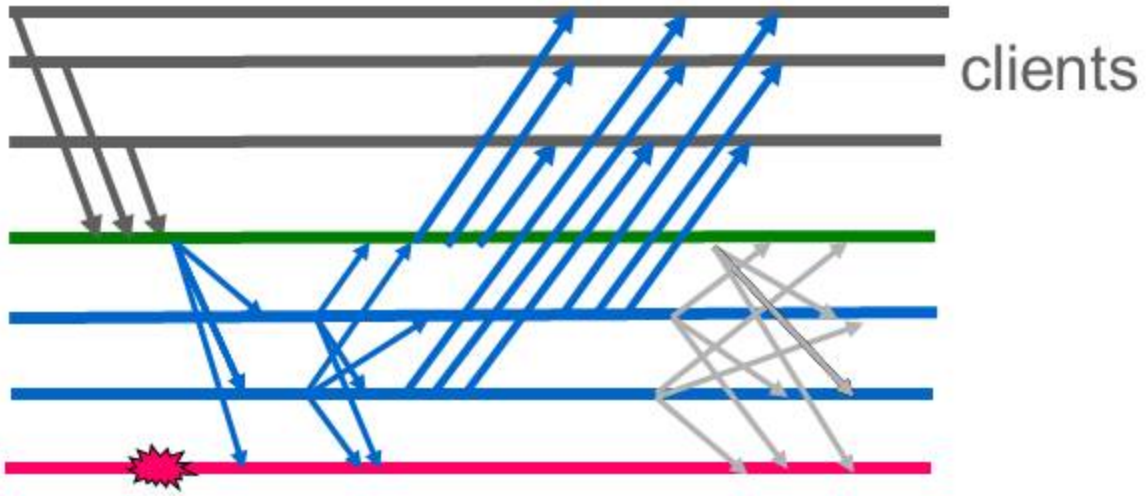
- Problems with reliable communication abstractions
  - communication can take arbitrarily long so
  - they need unlimited retransmission buffers
  - or stop when they run out of buffer space
- Solution in PBFT:
  - Receiver-based retransmissions

summary of k's state

- periodically multicast $\langle \text{STATUS}, v, h, e, P, C, k \rangle_{\alpha_k}$
- each replica i retransmits missing messages sent by i

Communication optimizations

- Batching:
  - run the algorithm on a batch of requests
  - amortizes cost across the batch
  - use sliding window to minimize impact on latency
  - significantly improves throughput



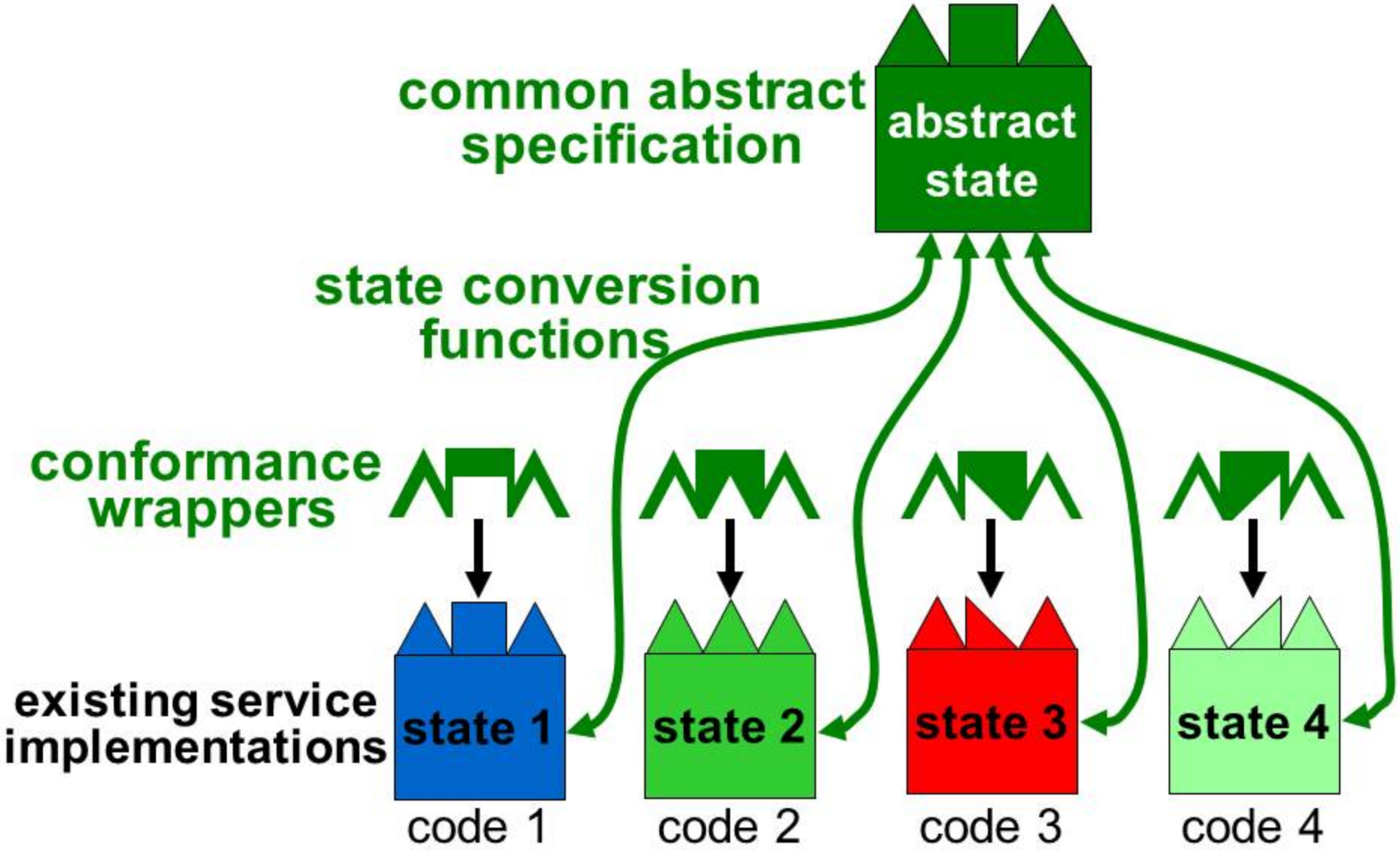
Reliable communication

- Problems with reliable communication abstractions
  - communication can take arbitrarily long so
  - they need unlimited retransmission buffers
  - or stop when they run out of buffer space
- Solution in PBFT:
  - Receiver-based retransmissions

summary of k's state

- periodically multicast $\langle \text{STATUS}, v, h, e, P, C, k \rangle_{\alpha_k}$
- each replica i retransmits missing messages sent by i

Abstraction methodology



Code Reuse: Single Implementation

- Methodology enables code reuse
  - less expensive to use Byzantine fault tolerance
  - existing implementation treated as black box
  - no modifications required

- Abstraction hides non-determinism
  - replicas can run non-deterministic code
  - replicas less likely to fail concurrently

Opportunistic N-Version Programming

- Run different off-the-shelf implementations
  - replicas less likely to fail concurrently
- Opportunistic N-Version programming is viable:
  - more than 4 implementations of important services
    - Example: file systems, databases
  - interoperability standards simplify methodology
    - Example: NFS, and ODBC

PBFT: client interface

```
int Byz_init_client(char* conf);

int Byz_invoke(Byz_req* req, Byz_rep* rep, bool read_only);
```

PBFT: server interface

int Byz_init_replica(char*conf, Upcall execute, Upcal get_obj, Upcall put_obj);

Upcall: int execute(Byz_req* req, Byz_rep* rep, int client_id, bool read_only);

// implement state conversion functions

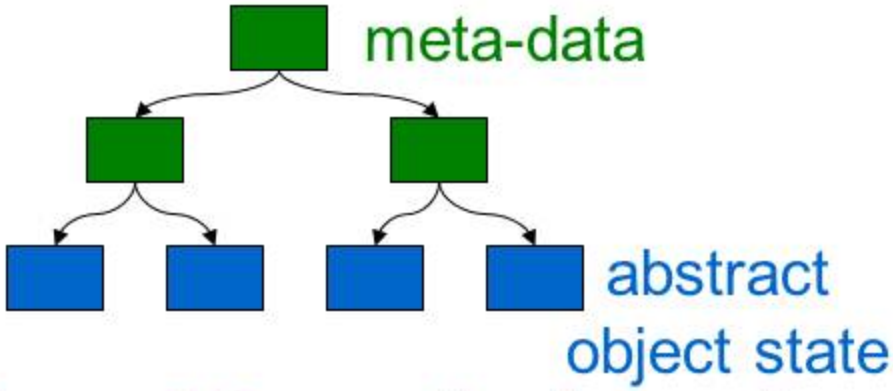
Upcall: int get_obj(int index, char** obj);

Upcall: void put_objs(int nobjs, char** objs, int * indices, int* sizes);

void Byz_modify(int index); // signal write to abstract object with given index

Checkpoints

- Efficient checkpoint creation:
  - periodic
  - copy-on-write
  - incremental digests with Merkle tree
- Secure and efficient state transfer:
  - hierarchical partitions
  - check chained digests

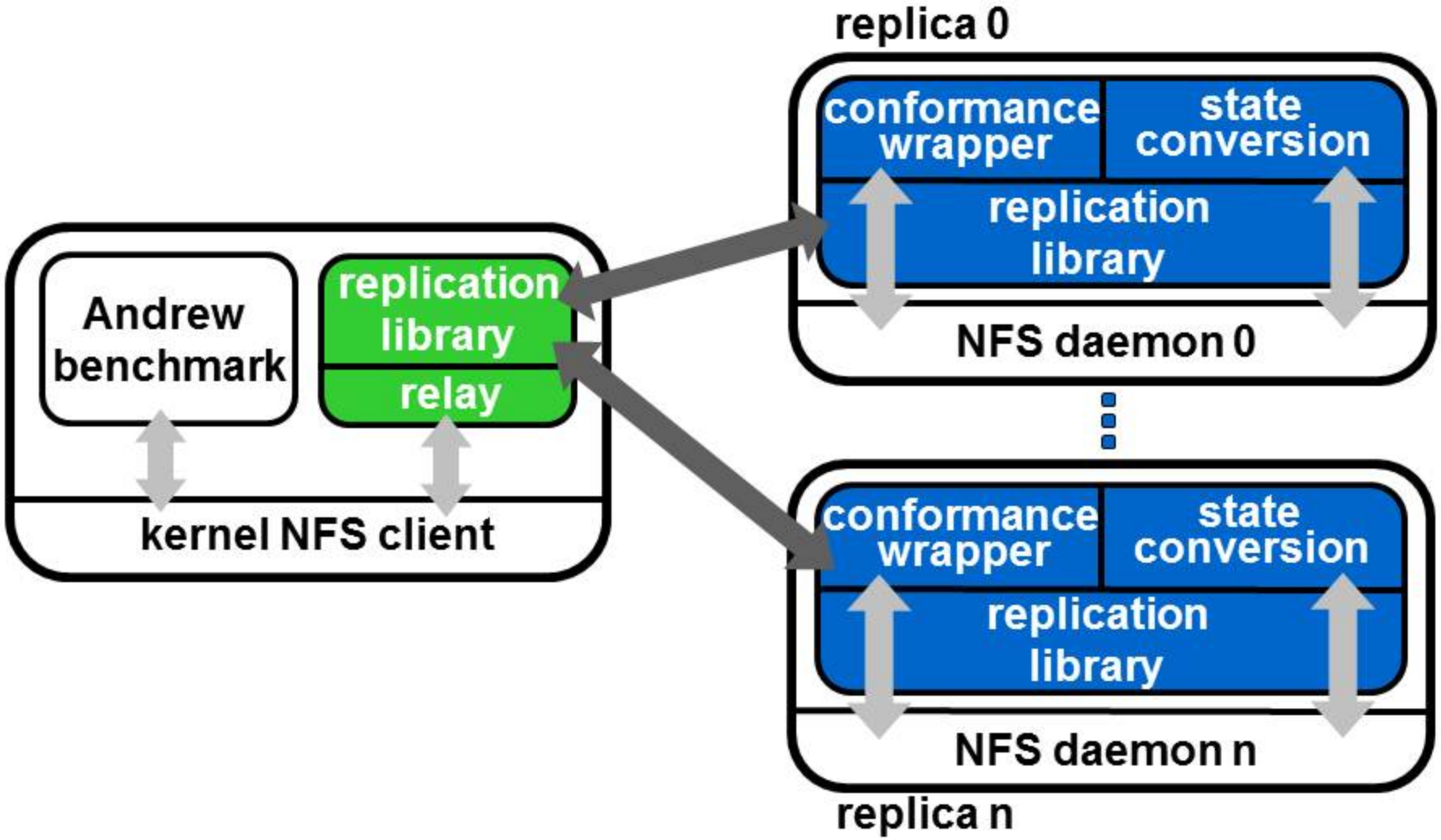


Fetch modified data only and from single replica

An Example: NFS File System

- Abstract specification based on NFS RFC
- Non-determinism problems in NFS:
  - file handle assignment
  - timestamp assignment
  - order of directory entries

NFS File System: Architecture



NFS File System: Wrapper

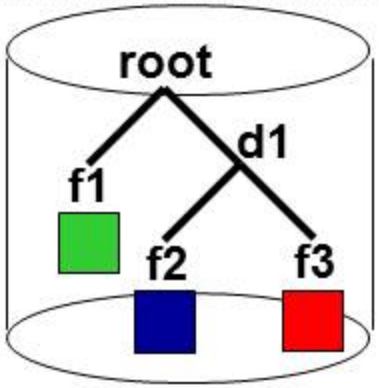
abstract state

	0	1	2	3	4	5
type	DIR	FILE	DIR	FILE	FILE	FREE
attributes	●	●	●	●	●	
contents	<f1,1> <d1,2>	■	<f2,3> <d3,4>	■	■	

wrapper state

	0	1	2	3	4	5
type	DIR	FILE	DIR	FILE	FILE	FREE
NFS file handle	h0	h1	h2	h3	h4	
timestamps	●	●	●	●	●	

concrete NFS server state



Evaluation

- Code complexity
  - simple code unlikely to introduce bugs
  - simple code costs less to write
- Overhead of wrapping and state conversions

Code Complexity

client relay	63
conformance wrapper	561
state conversions	481
total	1105

- Measured number of ;

NFS File System: Wrapper

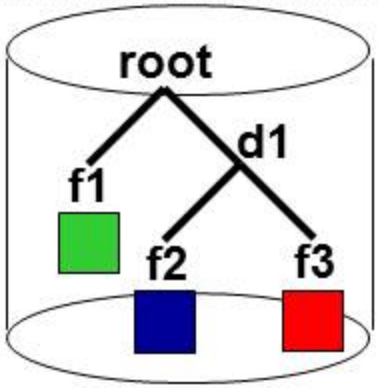
abstract state

	0	1	2	3	4	5
type	DIR	FILE	DIR	FILE	FILE	FREE
attributes	●	●	●	●	●	
contents	<f1,1> <d1,2>	■	<f2,3> <d3,4>	■	■	

wrapper state

	0	1	2	3	4	5
type	DIR	FILE	DIR	FILE	FILE	FREE
NFS file handle	h0	h1	h2	h3	h4	
timestamps	●	●	●	●	●	

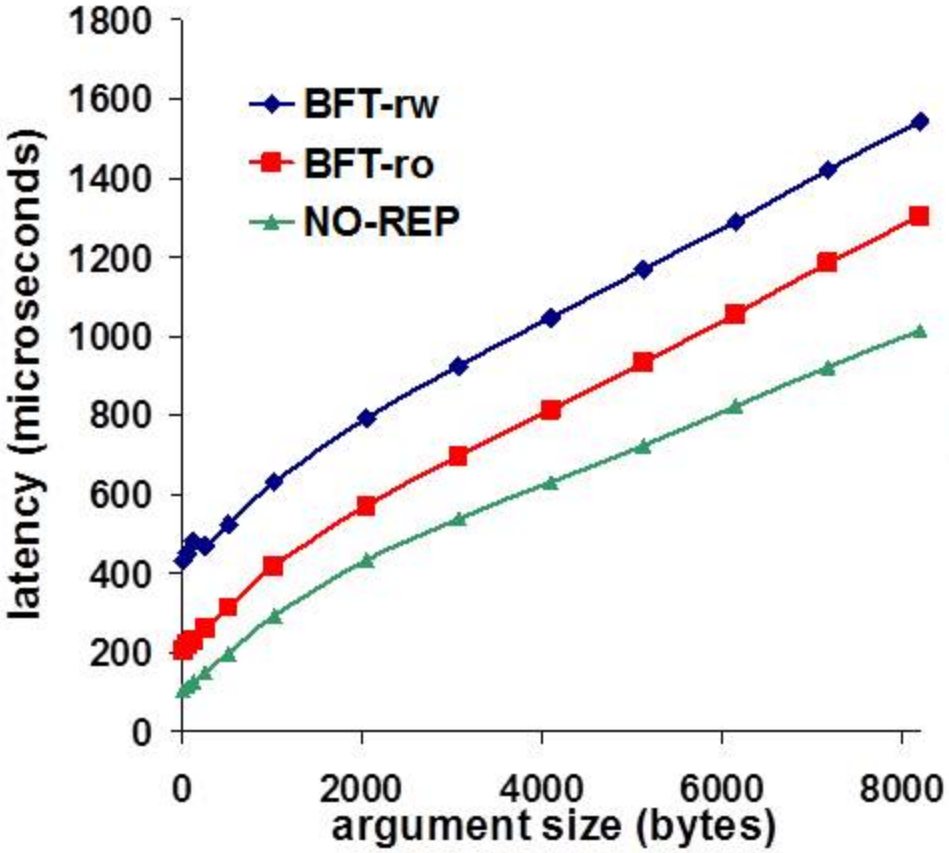
concrete NFS server state

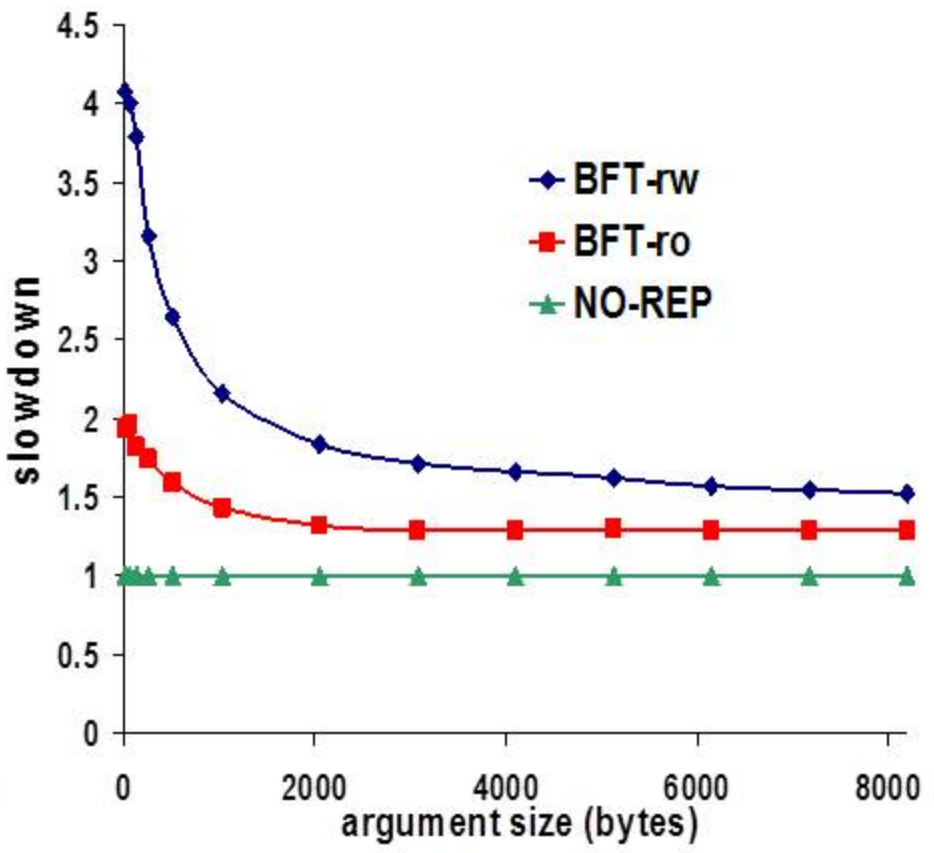


Performance Evaluation

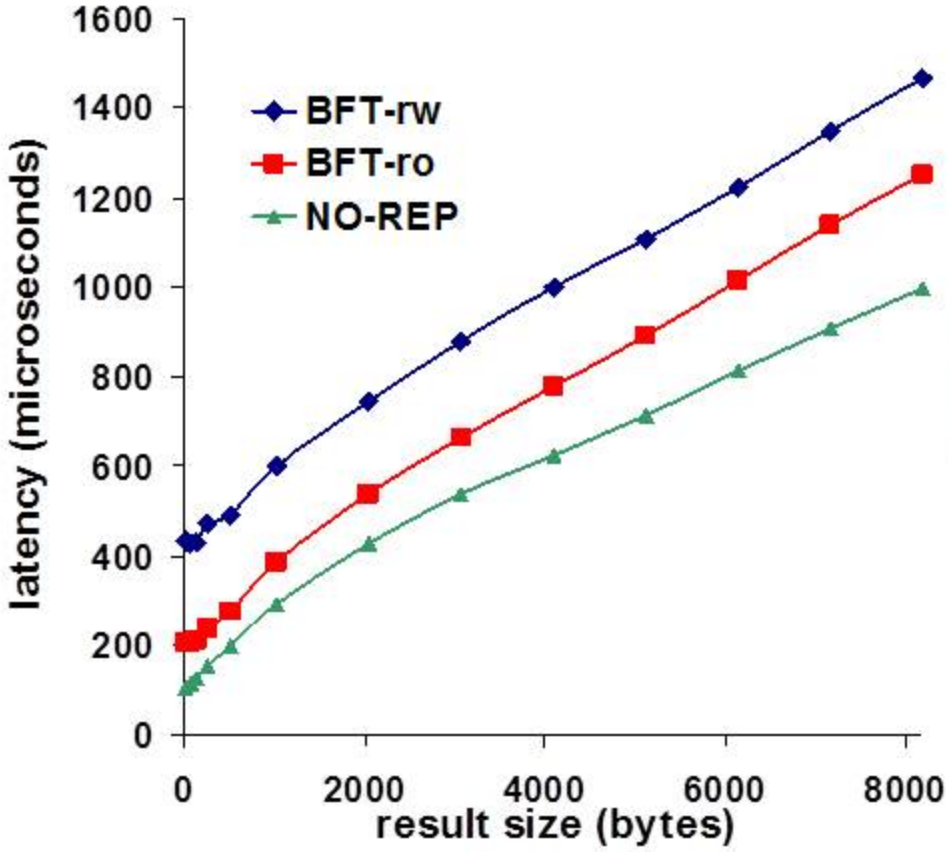
- What was measured:
  - time to complete recovery
  - maximum recovery frequency without overlap
  - overhead with maximum frequency
- Experimental setup:
  - MD5 digests
  - secret-suffix MD5 MACs with 16-byte keys
  - Rabin-Williams 1024-bit modulus
  - **1 client, 4 replicas** with Linux 2.2.16
  - Pentium III 600MHz, 512 MB RAM
  - Ethernet 100 Mbit/s

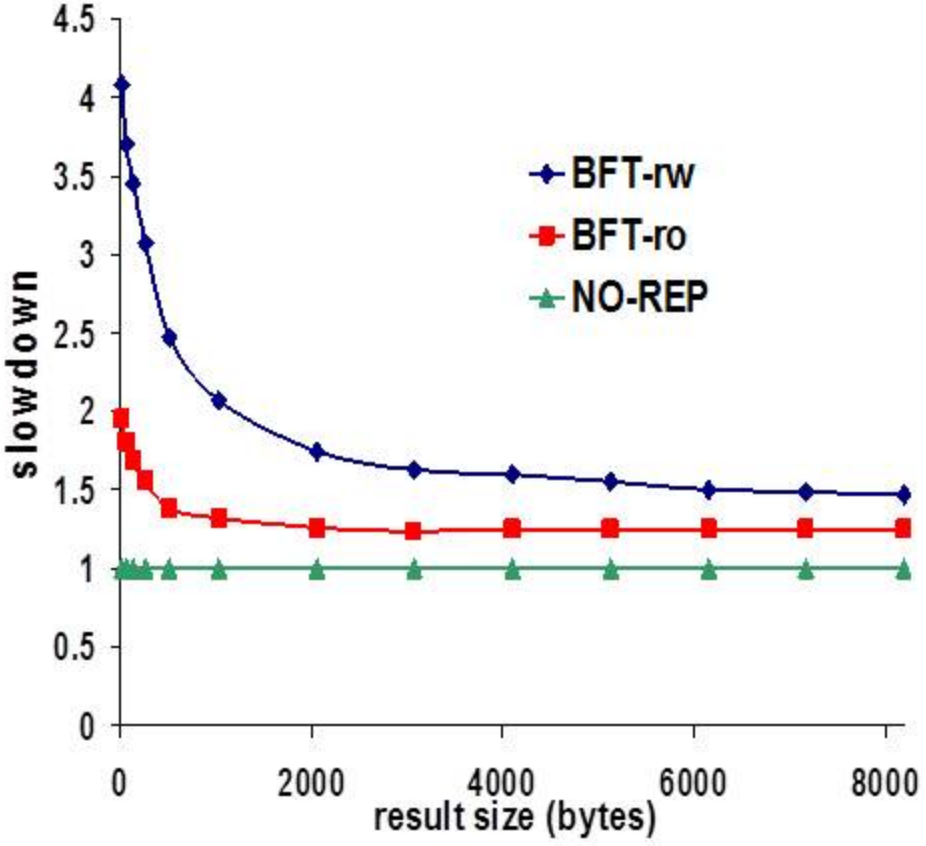
Latency With Varying Argument Size



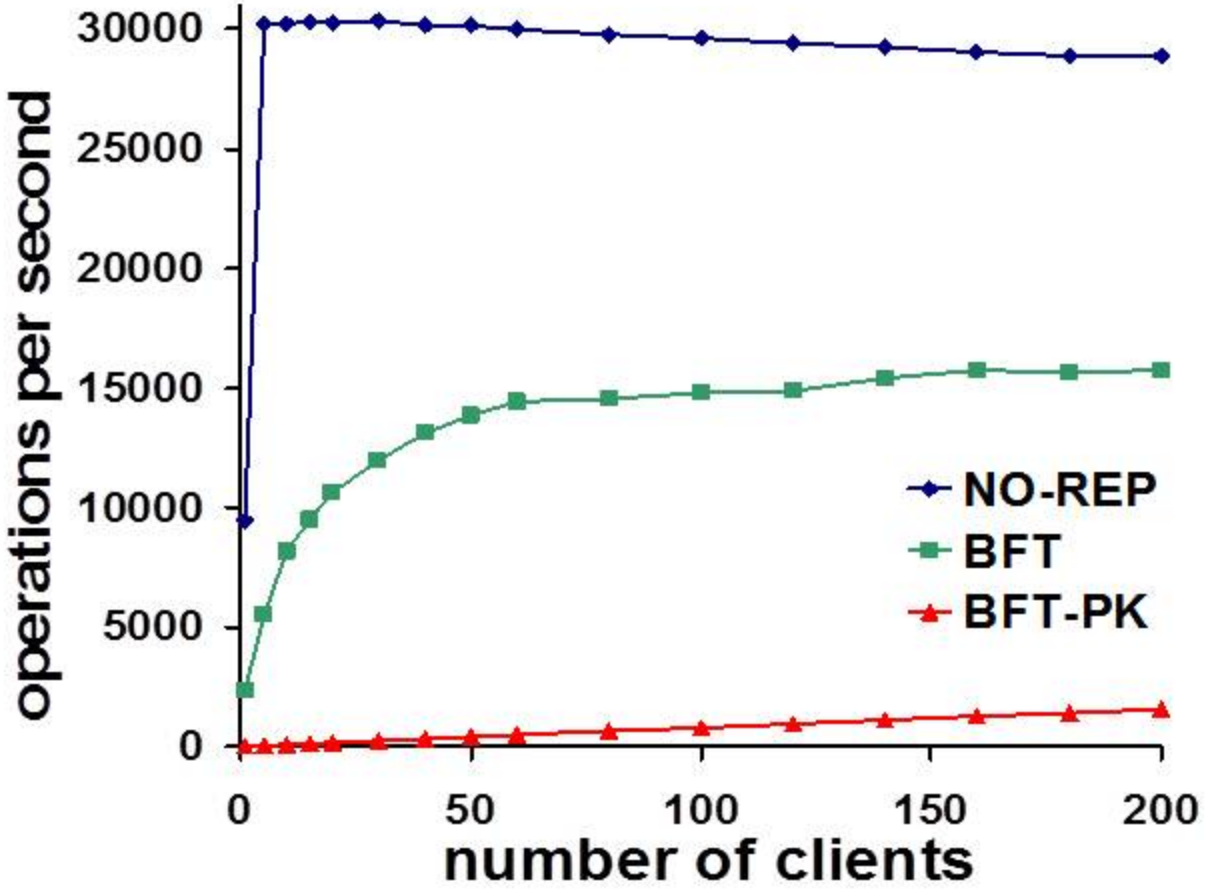


Latency With Varying Result Size

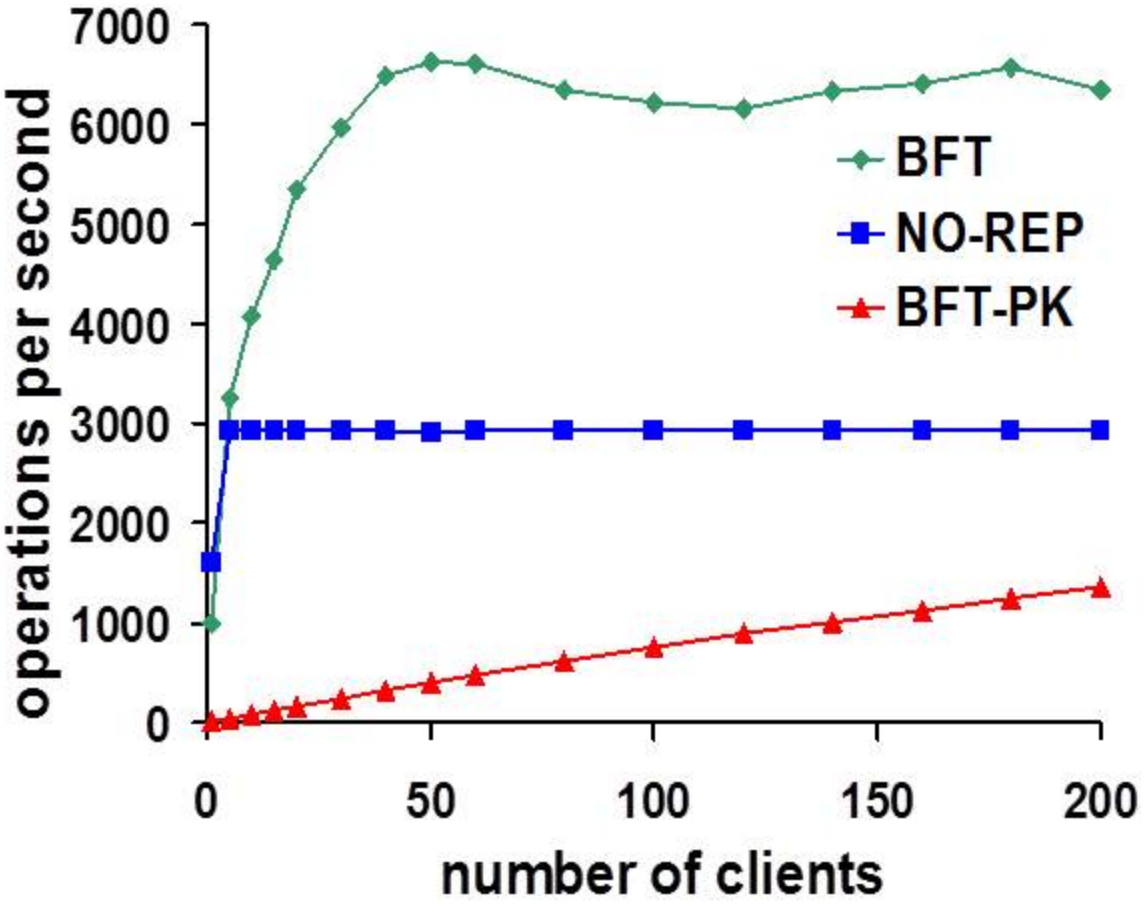




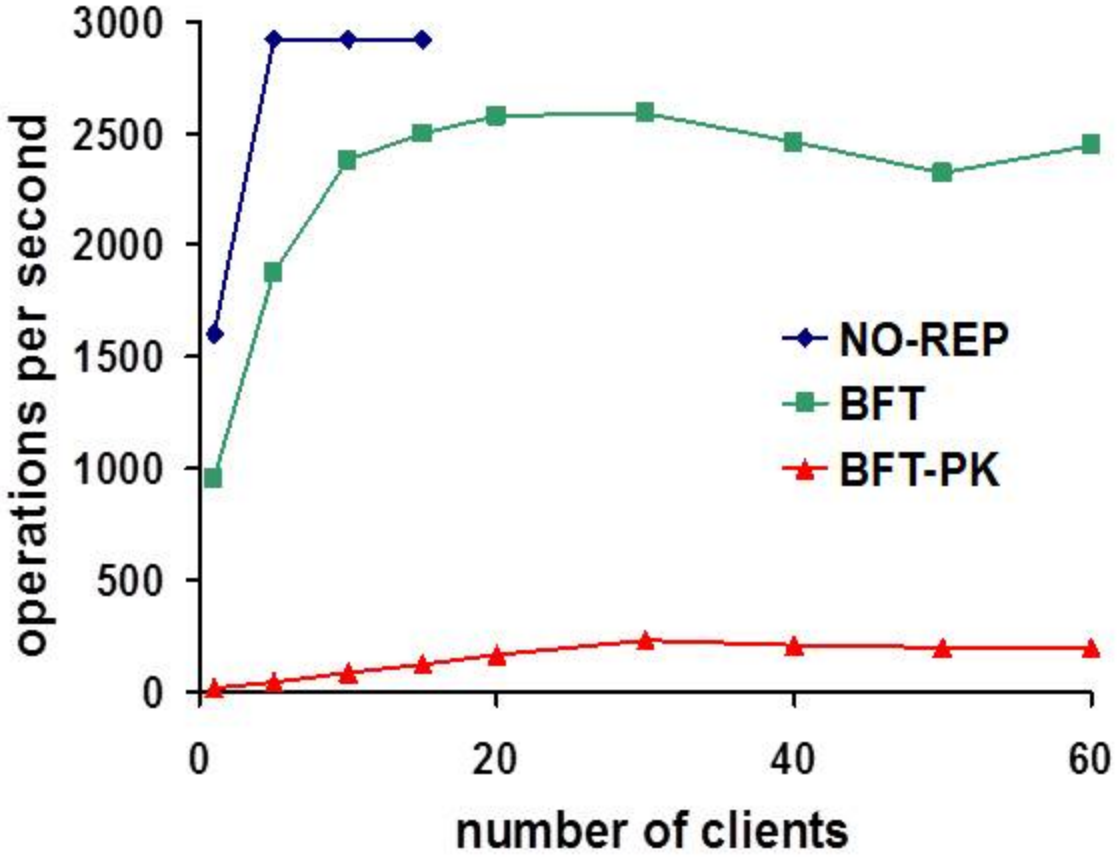
Throughput: 8B arguments and results



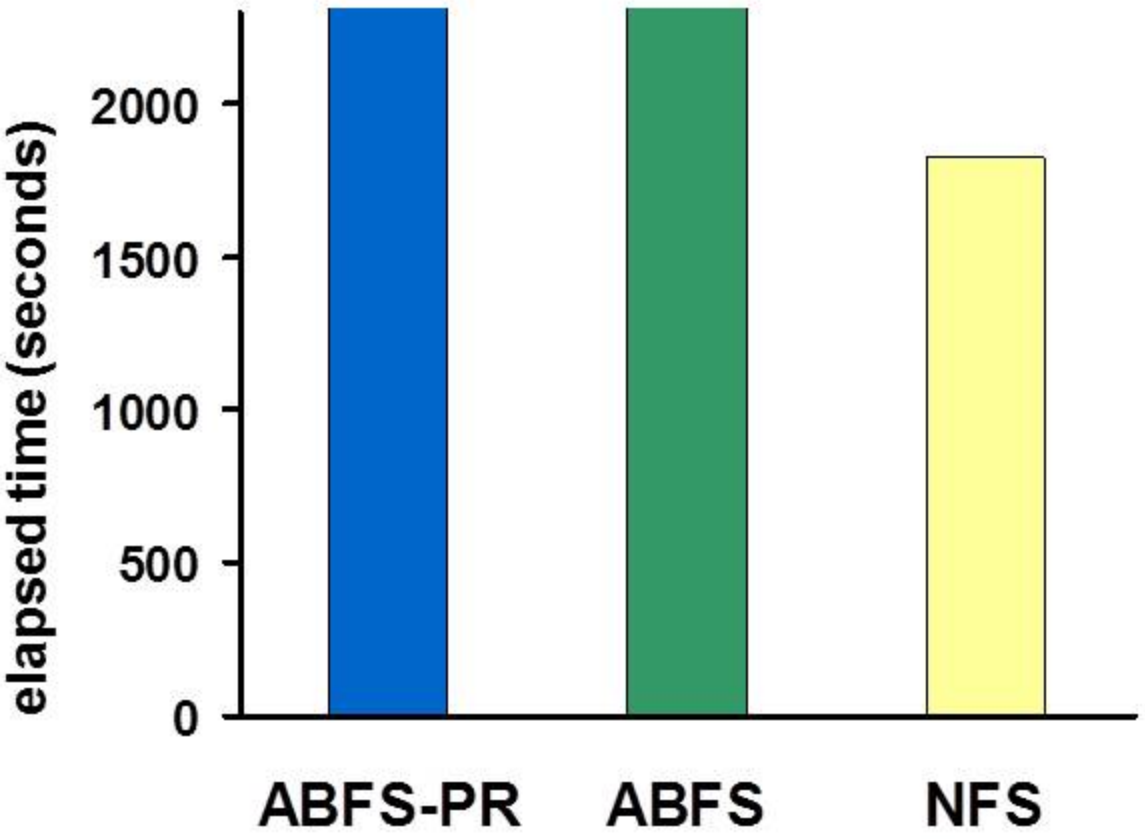
Throughput: 8B arguments, 4KB results



Throughput: 4KB arguments, 8B results



Overhead: Andrew500 (1GB)



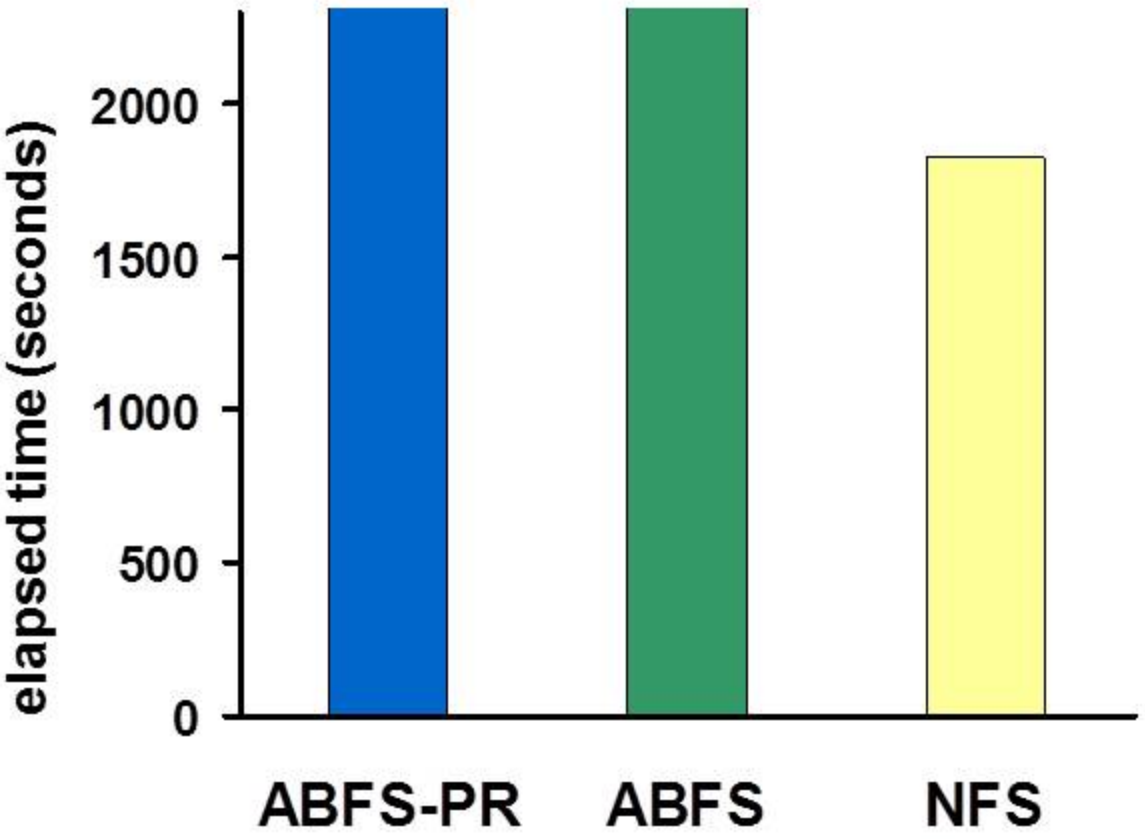
1 client, 4 replicas
Linux 2.2.16
Pentium III 600MHz
Fast Ethernet

- NFS is the NFS implementation in Linux
  - ABFS is our replicated file system
  - ABFS-PR has proactive recoveries

Overview

- Problem
- Assumptions
- Algorithm
- Implementation
- Performance
- Other BFT work

Overhead: Andrew500 (1GB)



1 client, 4 replicas
Linux 2.2.16
Pentium III 600MHz
Fast Ethernet

- NFS is the NFS implementation in Linux
  - ABFS is our replicated file system
  - ABFS-PR has proactive recoveries

Other BFT work

L. Lamport, R. Shostak, and M. Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982

The paper that coined the name Byzantine. It has algorithms for a synchronous system with and without signatures. Proves an upper bound of 3f+1 for algorithms without signatures.

Other BFT work

M. Reiter. The Rampart toolkit for building high-integrity services. *Theory and Practice in Distributed Systems (Lecture Notes in Computer Science 938)*, pages 99–110, 1995.

A system for state machine replication that uses group communication. Relies on synchrony assumptions for safety. Attacker can compromise system by delaying messages. Needs signatures.

Other BFT work

D. Malkhi and M. Reiter. Byzantine Quorum Systems. *Journal of Distributed Computing*, 11(4):203–213, 1998.

The best study of quorum systems that can tolerate Byzantine failures. Introduces a number of sophisticated constructions and analyzes their expected performance.

Other BFT work

C. Cachin, K. Kursawe, and V. Shoup. Random oracles in Constantinople: Practical asynchronous Byzantine agreement using cryptography. In *Proceedings of the 19th ACM Symposium on Principles of Distributed Computing,* July 2000.

A solution to asynchronous Byzantine agreement that uses randomization instead of eventual synchrony.

Other BFT work

Jinyuan Li, Maxwell N. Krohn, David Mazières, and Dennis Shasha. Secure Untrusted Data Repository (SUNDR). In Proceedings of the 6th Symposium on Operating Systems Design and Implementation, December 2004

System that provides secure storage on Byzantine faulty servers. Clients sign the full contents of the storage system each time they modify it. It uses signatures and Merkle trees to do this efficiently. Provides fork consistency a new consistency model weaker than linearizability.

Other BFT work

Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong, Zyzzyva: Speculative Byzantine Fault Tolerance, ACM Symposium on Operating Systems Principles (SOSP), October 2007

New state machine replication protocol that reduces delays and improves throughput by allowing replicas to execute requests speculatively after they receive a pre-prepare from the primary. It can achieve a throughput of nearly 80000 null request per second (PBFT can do about 60000 on the same hardware).

Other BFT work

A.Clement, M. Marchetti, E. Wong, L. Alvisi, and M. Dahlin, Making Byzantine Fault Tolerant Systems Tolerate Byzantine Faults, Symposium on Networked Systems Design and Implementation (NSDI), April 2009

Identifies several issues with the performance of previous systems under attack and proposes modification to achieve robust performance.

Conclusion

- Byzantine fault tolerance can be practical
- But no one is using it yet
- Added value does not justify overhead?
  - performance overhead is fairly low
  - but little added value without independent failures
  - checksums seem able to deal with most observed "Byzantine" failures