Practical Byzantine Fault Tolerance

Miguel Castro and Barbara Liskov

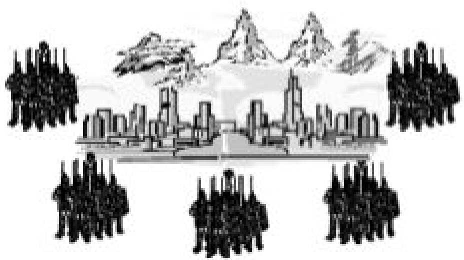
Outline

1. Introduction to Byzantine Fault Tolerance Problem
2. PBFT Algorithm
    a. Models and overview
    b. Three-phase protocol
    c. View-change
3. Implementation & Evaluation

Byzantine Fault Tolerance (BFT) Problem

Loi Luu

Historical Motivation*



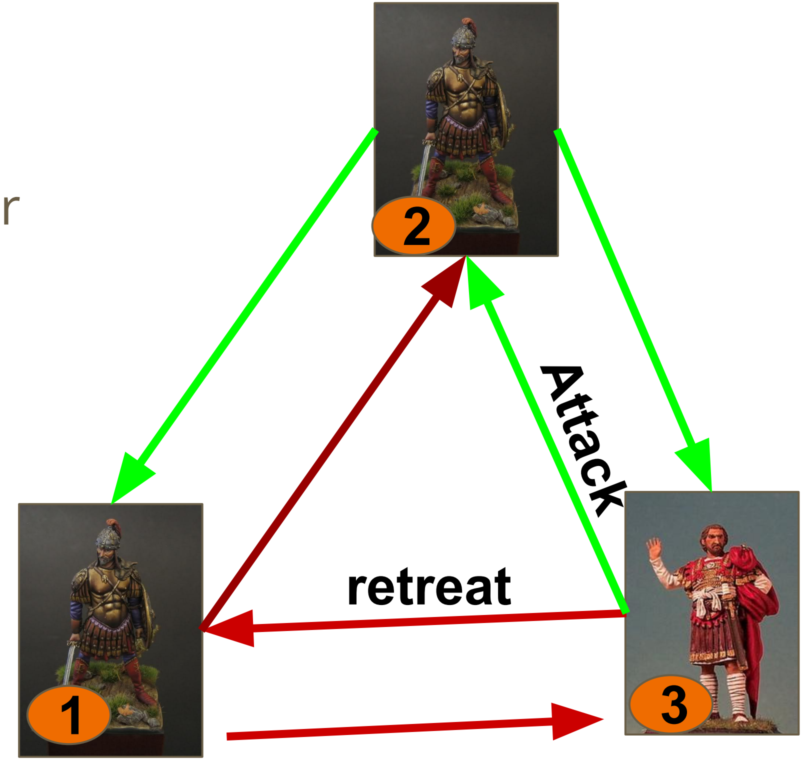
- A Byzantine army decides to attack/ retreat
  - **N** generals, **f** of them are traitors (can collude)
  - Generals camp outside the castle
    - Decide individually based on their field information
  - Exchange their plans by messengers
    - Can be killed, can be late, etc
- Requirements
  - All loyal generals agree on the same plan of action
  -

A BFT protocol helps loyal generals decide correctly

*http://research.microsoft.com/en-us/um/people/lamport/pubs/byz.pdf

Why is it hard?

- Simple scenario
  - 3 generals, third general is traitor
  - Traitor sends different plans
  - If decision is based on majority
    - (1) and (2) decide differently
    - (2) attacks and gets defeated
- More complicated scenarios
  - Messengers get killed, spoofed
  - Traitors confuse others:
    - (3) tells (1) that (2) retreats, etc



Computer Science Setting

- A general ⇔ a program component/ processor/ replica
  - Replicas communicate via messages/rpc calls
  - Traitors ⇔ Failed replicas
- Byzantine army ⇔ A deterministic replicated service
  - The service has states and some operations
  - The service should cope with failures
    - State should be consistent across replicas
  - Seen in many applications
    - replicated file systems, backup , Distributed servers
    - Shared ledger between banks

Byzantine Fault Tolerance Problem

- Distributed computing with faulty replicas
  - **N** replicas
  - **f** of them maybe faulty (crashed/ compromised)
  - Replicas initially start with the same state
- Given a request/ operation, the goal is:
  - Guarantee that all non-faulty replicas agree on the next state
  - Provide system consistency even when some replicas may be inconsistent

Properties

- Safety
  - *Agreement:* All non-faulty replicas agree on the same state
  - *Validity*: The chosen state is valid
- Liveness
  - Some state is eventually agreed
  - If a state has been chosen, all replicas eventually arrive at the state

1000+ Models of BFT Problem

- Network: synchronous, asynchronous, in between, etc
- Failure types: fail-stop (crash), Byzantine, etc
- Adversarial model
  - Computationally bounded
  - Universal adversary: can see everything, private channels
  - Static, dynamic adversary
- Communication types
  - Message passing, broadcast, shared registers
- Identities of replicas
  - Pre-established/ unknown?

An algorithm that works for one model may not work for others!

- Sparse network, full (complete) network

Previous Work

- The "celebrated" Impossibility Result
  - Only one faulty replica makes (*deterministic*) agreement impossible in the asynchronous model
  - Intuition
    - A faulty replica may just be slow, and vice versa.
    - E.g. cannot make progress if don't receive enough messages
  - Most protocols
    - Require synchrony assumption to achieve safety and liveness
    - Have some *randomization*: terminate with high prob., agreement can be altered with non-zero prob., etc.

Previous Work(2)

- Paxos
  - Model
    - Network is asynchronous (messages are delayed arbitrarily, but eventually delivered)
    - Tolerate crashed failure
  - Guarantee safety, but not liveness
    - The protocol may not terminate
    - Terminate if the network is synchronous eventually
  - One of the main results
    - Require at least **3f+1** replicas to tolerate **f** faulty replicas

Is Crashed Failure Good Enough?

- Byzantine failures are on the rise
  - Malicious successful attacks become more serious
  - Software errors are more due to the growth in size and complexity of software
  - Faulty replicas exhibit Byzantine behaviors
- How to reach agreement even with Byzantine failures?

Practical Byzantine Fault Tolerance*

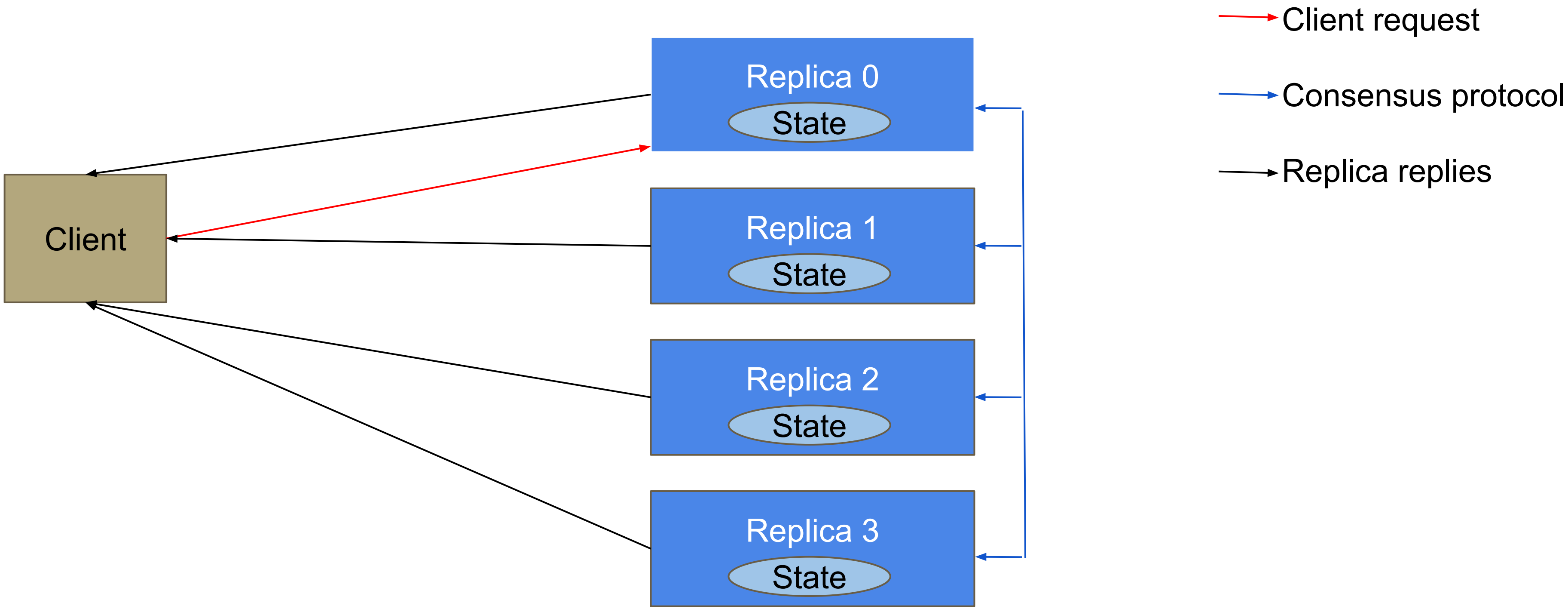
- Is introduced almost 20 years after Paxos
- Model in PBFT is practical
  - Asynchronous network
  - Byzantine failure
- Performance is better
  - Low overhead, can run in real applications
- Adoption in industry
  - See Tendermint, IBM's Openchain, and ErisDB

*http://pmg.csail.mit.edu/papers/osdi99.pdf

PBFT Algorithm

Hung Dang

System Model



System Model

- Asynchronous distributed system
    - Delay*, duplicate or deliver messages out of order
- Byzantine failure model
    - Faulty replicas may behave arbitrarily
- Preventing spoofing and relays and corrupting messages
    - Public-key signature: one cannot impersonate other
    - Message authentication code, collision-resistant hash: one cannot tamper other's messages

* Messages are delivered *eventually*

Adversary Model

- Can coordinate faulty replicas
- Delay communications, but not indefinitely
- Cannot subvert the cryptographic techniques employed

Service Properties

- Safety
- Liveness
- Optimal resiliency
  - To tolerate f faulty replicas, the system requires $\boldsymbol{n = 3f+1}$ replicas
  - Can proceed after communicating with $n - f$ (i.e. $2f+1$) replicas:
    - If none of those $2f+1$ replicas is faulty, good
    - Even if up to f of them are faulty, the other $f+1$ (i.e. the majority) are not => ensure safety

The Algorithm

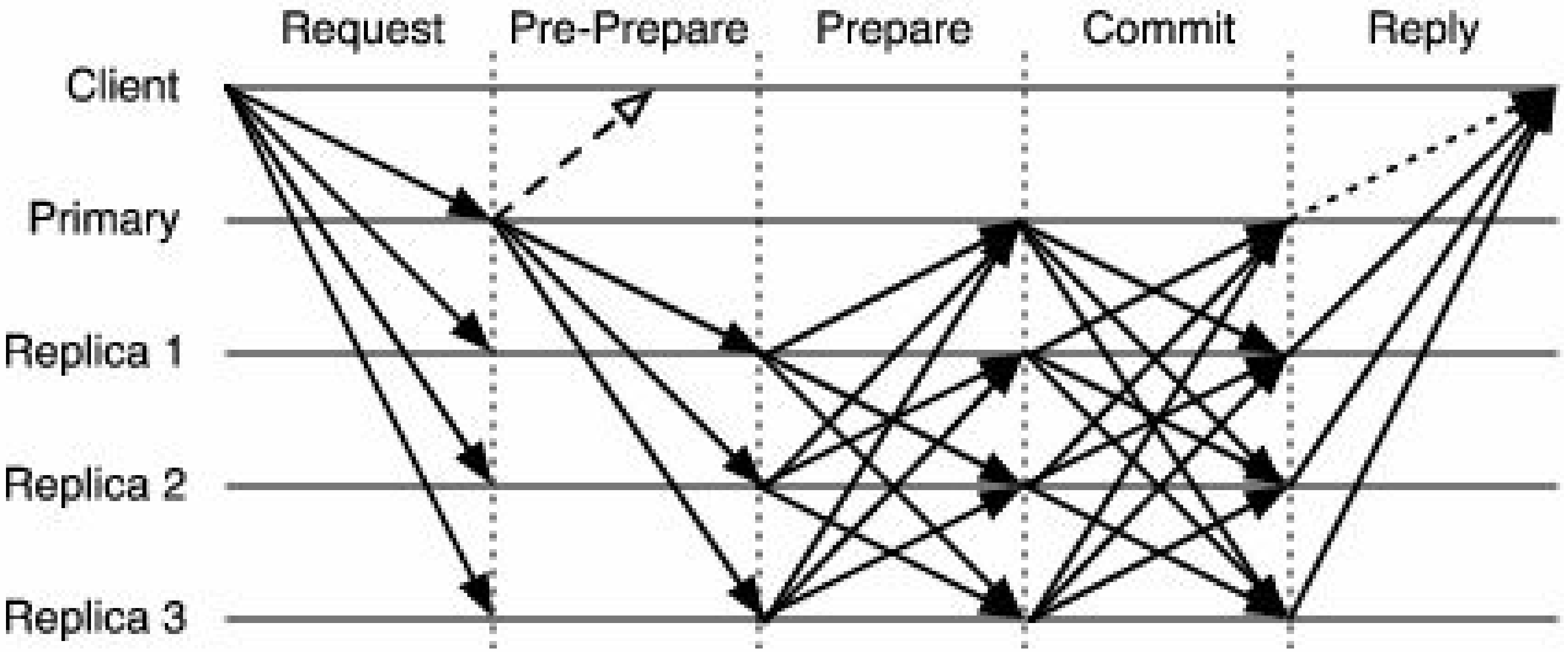
- The set of replica is R; $|R| = 3f+1$ (f is # of faulty replicas tolerated)
- Each replica is identified by an integer in $\{0,...,3f\}$
- Each replica is deterministic and starts at the same initial state
- A view is a configuration of replicas:
  - replica $p = v \bmod |R|$ is the *primary* of view v
  - all other replicas are *backups*

The Algorithm

1. Client sends request* to the primary.
2. Primary validates the request and initiates the 3-phase protocol (pre-prepare → prepare → commit) to ensure consensus among all (non-faulty) replicas.
3. The replicas execute the request and send result directly to the client.
4. The client accepts the result after receiving f+1 identical replies.

* It is assumed that the client waits for one request to complete before sending the next one

The Algorithm



The rationale of the three-phase protocol

Divya Sivasankaran

Three Phase Protocol - Goals

Ensure safety and liveness despite asynchronous nature

- Establish total order of execution of requests (*Pre-prepare* + *Prepare*)
- Ensure requests are ordered consistently across views (*Commit*)

Recall: View is a configuration of replicas with a primary $p = v \bmod |R|$

REQUEST → ***PRE-PREPARE*** → ***PREPARE*** → ***COMMIT*** → REPLY

Three Phases:

- Pre-prepare
  - Acknowledge a ***unique sequence number*** for the request
- Prepare
  - The replicas agree on this sequence number
- Commit
  - Establish total order across views

REQUEST → ***PRE-PREPARE*** → ***PREPARE*** → ***COMMIT*** → REPLY

Definitions

- Request message m
- Sequence number n
- Signature - σ
- View - v
- Primary replica - p
- Digest of message D(m) $\rightarrow$ d

Pre-prepare

Purpose: acknowledge a unique sequence number for the request

- SEND
  - The primary assigns the request a sequence number and broadcasts this to all replicas

- A backup will ACCEPT the message iff
  - d, v, n, σ are valid
  - (v,n) has not been processed before for another digest (d)

REQUEST → ***PRE-PREPARE*** → PREPARE → COMMIT → REPLY

Prepare

Purpose: The replicas agree on this sequence number

After backup i accepts <PRE-PREPARE> message

- SEND
  - multicast a <PREPARE> message acknowledging n, d, i and v

- A replica will ACCEPT the message iff
  - d, v, n, σ are valid

REQUEST → PRE-PREPARE → ***PREPARE*** → COMMIT → REPLY

Prepared

Predicate prepared(m,v,n,i) = T iff replica i

- <PRE-PREPARE> for m has been received
- **2f+1**(incl itself) distinct & valid <PREPARE> messages received

Guarantee

Two **different** messages can never have the same sequence number

i.e., *Non-faulty replicas agree on total order for requests within a view*

REQUEST → PRE-PREPARE → ***PREPARE*** → COMMIT → REPLY

Commit

Purpose: Establish total order across views

Once prepared(m,v,n,i) = T for a replica i

- Send
  - multicast <COMMIT> message to all replicas

- All replicas ACCEPT the message iff
  - d, v, n, σ are valid

REQUEST → PRE-PREPARE → PREPARE → ***COMMIT*** → REPLY

Committed

Predicate committed(m,v,n,i) = T iff replica i

- prepared(m,v,n,i) = T
- **2f+1**(incl itself) distinct & valid <COMMIT> messages received

Guarantee

Total ordering across views (*Proof will be shown later*)

REQUEST → PRE-PREPARE → PREPARE → ***COMMIT*** → REPLY

Executing Requests

Replica i executes request iff

- committed(m,v,n,i) = T
- All requests with lower seq# are already executed

Once executed, the replicas will directly send <REPLY> to the client

But, what if the primary is faulty? How can we ensure the system will recover?

REQUEST → PRE-PREPARE → PREPARE → COMMIT → ***REPLY***

View Change

Irvan

View Change

All is good if primary is good

But everything changed when primary is faulty...

Problem (Case 1)

Sequence number 1: **INSERT (APPLE) INTO FRUIT**

Sequence number 4: **INSERT (PEAR) INTO FRUIT**

Sequence number 5: **SELECT * FROM FRUIT**

The replica will be stuck waiting for request with sequence number 2…

View Change Idea

- Whenever a lot of non-faulty replicas detect that the primary is faulty, they together begin the *view-change operation*.
    - More specifically, if they are stuck, they will suspect that the primary is faulty
    - The primary is detected to be faulty by using timeout
    - **Thus this part depends on the synchrony assumption**
    - They will then change the view
        - The primary will change from replica p to replica (p+1)%|R|

Initiating View Change

- Every replica that wants to begin a view change sends a <VIEW-CHANGE> message to EVERYONE
    - Includes the current state so that all replicas will know which requests haven't been committed yet (due to faulty primary).
    - List of requests that was **prepared**
- When the new primary receives **2f+1** <VIEW-CHANGE> messages, it will begin the view change

The Corresponding Message

Sequence number 1: **INSERT (APPLE) INTO FRUIT**

Sequence number 4: **INSERT (PEAR) INTO FRUIT**

Sequence number 5: **SELECT * FROM FRUIT**

Replica 1 <VIEW-CHANGE> message:

<VIEW-CHANGE, **SEQ1: INSERT (APPLE)**, **SEQ4: INSERT (PEAR)**, **SEQ5: SELECT ***>

View-Change and Correctness

1) New primary gathers information about which requests that need committing

- This information is included in the <VIEW-CHANGE> message
- All replicas can also compute this since they also receive the <VIEW-CHANGE> message
  - Will avoid a faulty new primary making the state inconsistent

2) New primary sends <NEW-VIEW> to all replicas

3) All replicas perform 3 phases on all the requests again

Example

<VIEW-CHANGE, **SEQ1: INSERT (APPLE)**, **SEQ4: INSERT (PEAR)**, **SEQ5: SELECT ***>
<VIEW-CHANGE, **SEQ2: INSERT (KIWI)**, **SEQ4: INSERT (PEAR)**, **SEQ5: SELECT ***>

Sequence number 1: **INSERT (APPLE) INTO FRUIT**
Sequence number 2: **INSERT (KIWI) INTO FRUIT**

Sequence number 4: **INSERT (PEAR) INTO FRUIT**
Sequence number 5: **SELECT * FROM FRUIT**

...Will still get stuck on sequence number 3?

Example

<VIEW-CHANGE, **SEQ1: INSERT (APPLE)**, **SEQ4: INSERT (PEAR)**, **SEQ5: SELECT ***>
<VIEW-CHANGE, **SEQ2: INSERT (KIWI)**, **SEQ4: INSERT (PEAR)**, **SEQ5: SELECT ***>

Sequence number 1: **INSERT (APPLE) INTO FRUIT**
Sequence number 2: **INSERT (KIWI) INTO FRUIT**
Sequence number 3: PASS
Sequence number 4: **INSERT (PEAR) INTO FRUIT**
Sequence number 5: **SELECT * FROM FRUIT**

Sequence numbers with missing requests are replaced with a “no-op” operation - a “fake” operation.

State Recomputation

- Recall the new primary needs to recompute which requests need to be committed again.
- Redoing all the requests is expensive
- Use checkpoints to speed up the process
    - After every 100 sequence number, all replicas save its current state into a checkpoint
    - Replicas should agree on the checkpoints as well.

Other types of problems...

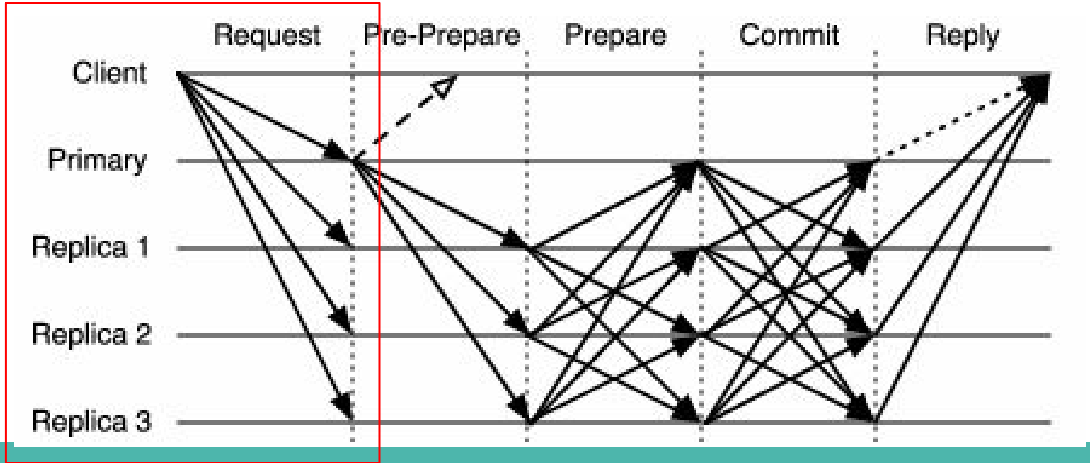
- What happens if the new primary is also faulty?
  - Use another timeout in the view-change
    - When the timeout expires, another replica will be chosen as primary
    - Since there are at most f faulty replicas, the primary can be consecutively faulty for at most f times
- What happen if a faulty primary picks a huge sequence number? For example, 10,000,000,000?
  - The sequence number must lie within a certain interval
  - This interval will be updated periodically

Problem (Case 2)

- Client sends request to primary
- Primary doesn’t forward the request to the replicas…

Client Full Protocol

- Client sends a request to the primary that they knew
  - The primary may already change, this will be handled
- If they do not receive reply within a period of time, it broadcast the request to all replicas



Replica Protocol

- If a replica receive a request from a client but not from the primary, they send the request to the primary,
- If they still do not receive reply from primary within a period of time, they begin view-change

Some Correctness

To convince you that the view-change protocol preserves safety, we will show you one of the key proofs

Correctness of View-Change

- We will show that if at any moment a replica has **committed** a request, then this request will ALWAYS be re-committed in the view-change

Proof Sketch

- Recall that a request will be re-committed in the view-change if they are included in at least one of the <VIEW-CHANGE> messages
- A **committed** request implies there are at least f+1 non-faulty replicas that ***prepared*** it.
- Proof:
  - There are 2f+1 <VIEW-CHANGE> messages
  - For any request **m** that has been committed, there are f+1 non-faulty replicas that ***prepared*** **m**
  - Since |R| = 3f+1, <u>at least one non-faulty replicas mu</u> <u>prepared **m** and sent the <VIEW-CHANGE> message</u>

Notes

- This safety lemma is one of the reasons we need to have a three phase protocol instead of two phase protocols
  - In particular, if we only have two phases, we cannot guarantee that if a request has been committed, it will be prepared by a majority of non-faulty replicas. Thus it’s possible that an committed request will not be re-committed... -- violates safety.

Optimization, Implementation and Evaluation

Zheyuan Gao

Optimization

- Reduce the cost of communication
- Reduce message delays
- Improve the performance read-only operations
- ……

Reduce the Cost of Communication

- A client designates one replica to send the full result.
- All other replicas send replies containing just the **digest** of the result, which allows client:
  - Check the correctness of the result.
  - Reduce network bandwidth consumption and CPU overhead.
- If client doesn’t receive enough valid digests, it retransmits the request asking all replicas to send the result.
- Original method requires all the replicas to send the full result, now only requires one replica to send the result, others just send the digest of the result.

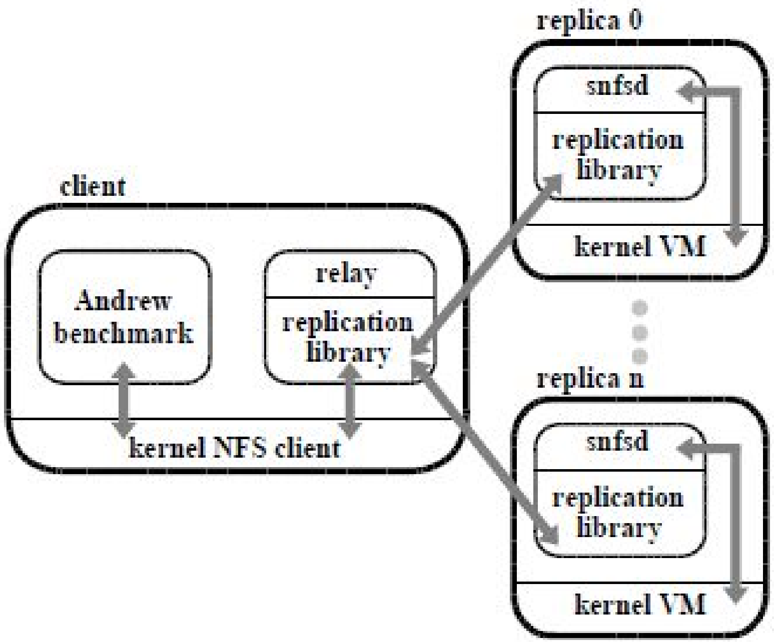
Reduce the Message Delays

- Replicas execute a request *tentatively* after
  - After receiving 2f+1 prepare messages, execute it *tentatively*.
- The client waits for 2f+1 matching tentative replies to guarantee that these replicas will commit eventually. Otherwise, the client retransmits the request and waits for f+1 non-tentative replies.
- In original implementation the PBFT requires 5 steps to detect whether the replied result is valid or not, now it only requires 4 steps(By judging the tentative replies).

Improve the Performance Read-only Operations

- A client multicasts a read-only request to all replicas.
- Replicas execute the request after:
  - Checking the request is authenticated (Client has access).
  - The request is in fact read-only.
- Replicas send back a reply only after all requests it executed before the read-only request have committed.
- Clients waits for 2f+1 replies from different replicas with same result.
- This reduces latency to a single round trip for most read-only requests.

BFS: A Byzantine-Fault-tolerant File System



Performance Evaluation

- A micro-benchmark
  - Provides service-independent evaluation of the replication library(Latency of invocation)
- Andrew benchmark
  - Compare BFS with two other file systems.
  - Allow us to evaluate the overhead of this algorithm accurately within an implementation of a real service.

Micro-Benchmark

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.

Andrew Benchmark

phase	BFS strict	BFS r/o lookup	BFS-nr
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.

Andrew Benchmark

phase	BFS strict	BFS r/o lookup	NFS-std
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.

NFS:Network File System

Summary

1. Introduction to Byzantine Fault Tolerance Problem
2. PBFT Algorithm
    a. Models and overview
    b. Three-phase protocol
    c. View-change
3. Implementation & Evaluation

Thank you!

A Variant of BFT: Byzantine General Problem

- One replica is primary, others are backups
  - Replicas know who is the current primary
- Primary replica sends operations to others
- Properties
  - Safety
    - Replicas agree on the next state, otherwise detect the primary is faulty
  - Liveness
    - Faulty replicas cannot block the system forever