

Practical Byzantine Fault Tolerance

Miguel Castro and Barbra Liskov (Turing Award Winner)

Presented by Aaron Schulman

Why isn't Naive BFT **Practical**?

- Strong assumptions
 - Synchrony
- Requires message flooding

What does the system guarantee?

- **Liveness** and **safety** with at most faulty replicas (out of n)

$$\left\lfloor \frac{n-1}{3} \right\rfloor$$

Starting with **weaker** assumptions

- Messages:
 - Asynchronous
 - Dropped, Duplicated, Delayed (IP)
 - Authenticated
 - Signed
- Faulty nodes may behave arbitrarily

Still a few **strong** assumptions

- Independent node failures
 - n nodes, n different versions of code
- Nodes are deterministic
 - Given X always return Y
- Bounded # of faulty nodes f

Assumptions about faulty nodes

- Bounded number f
- $3f+1$ minimum number of replicas for safety and liveness
- non-faulty have to outnumber faulty
 - $(n-2f > f)$ so $(n > 3f)$
- non-faulty nodes see them the same way

Adversarial model

- Coordinate faulty nodes
- Delay communication (correct or faulty)
 - Not indefinitely
- Can't break crypto (signature or digest)
- All clients are seen consistently

Node types in the BFT system

- Client - Makes request
- Primary - Receives client's request
- Backup (Replica) - Executes request

Client

- Request has the following:
- Timestamp increasing for each request (Seq#)
 - Client #
 - Operation

Primary

- Each view has a primary
- selection is
 - **primary # = view # mod list of replicas**

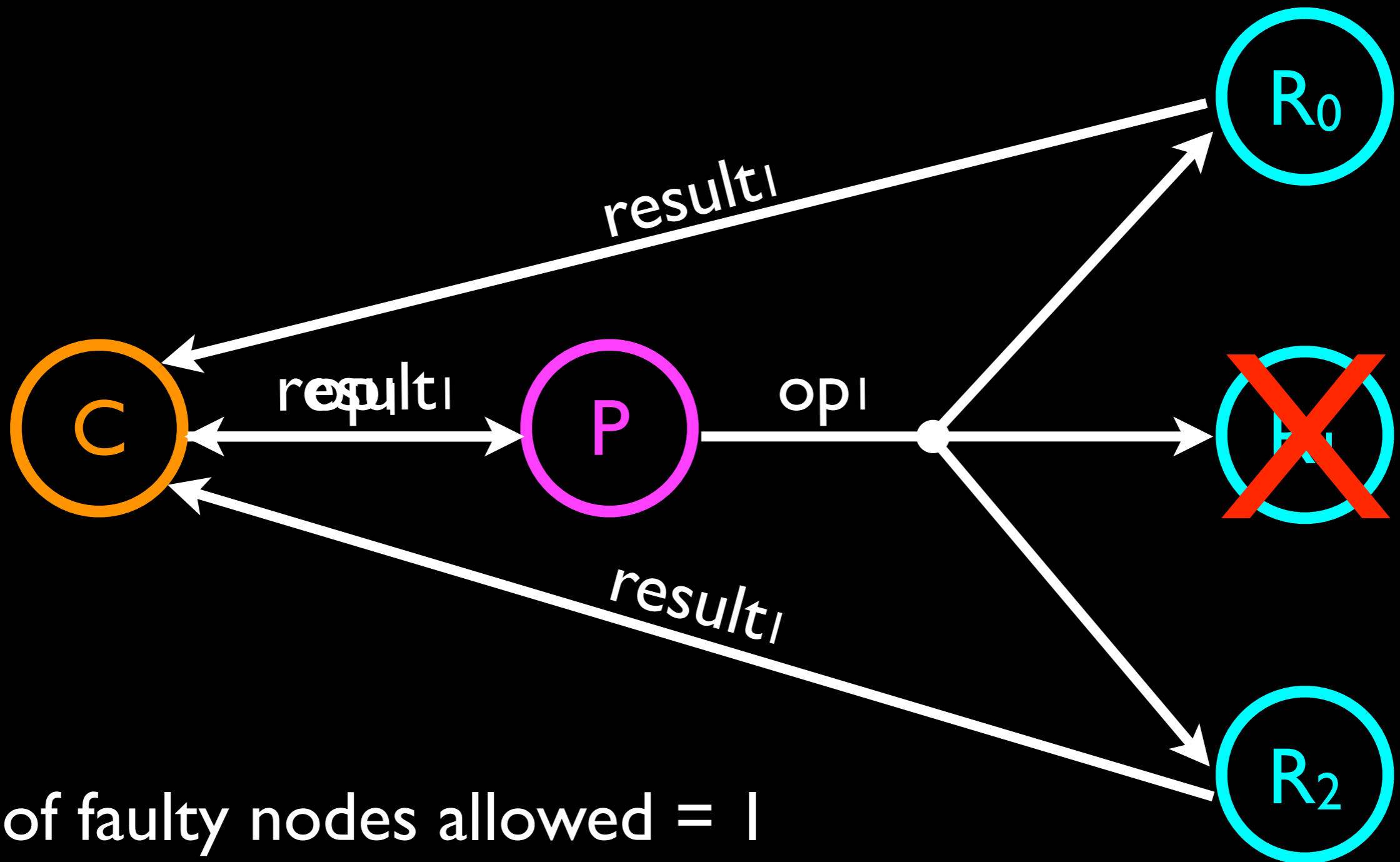
View

- State of replicated data and primary
- Replicas move through a succession of views
- Views change when a primary is faulty

Backup (replica)

- State:
 - Message log - messages can be out of order
 - Last stable checkpoint, non-stable checkpoint, current state
- View #

The high level algorithm



of faulty nodes allowed = 1

But of course it is not that
simple...

There are no ordering
guarantees for messages

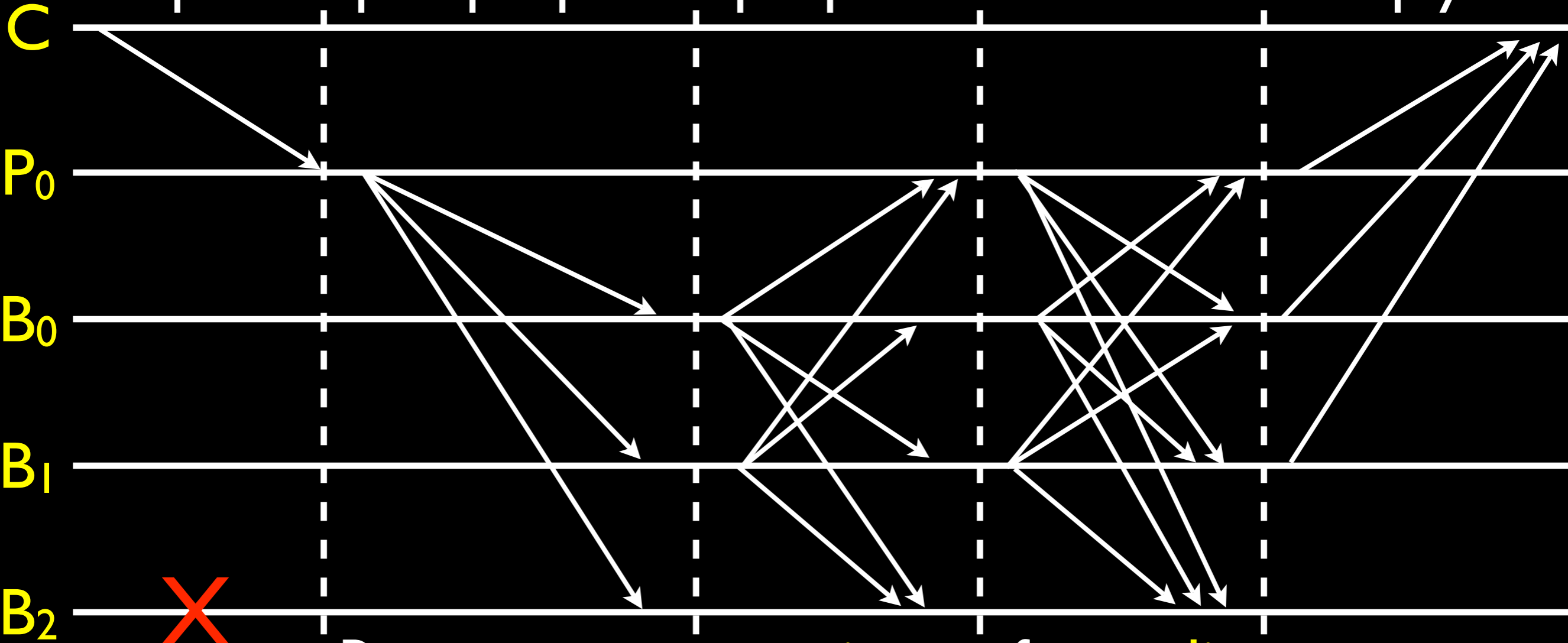
Accept message if...

- Signatures are correct
- Digests are correct (request, view change)
- For the correct view
- Has not accepted a message of that type with a lower sequence number
- The sequence number is between the low and high watermark

ensure ordering
1 in a view 3

ensure ordering
in 6 multiple views 9 3

request | pre-prepare | prepare | commit | reply



Request sent to **primary** from **client**
Log, assign seq. #, and multicast to **all backups**
Multicast to **all backups**
Multicast to **all backups**

Removing messages from the log

- Checkpointing:
 - Only continue when $f + 1$ replicas send reply
 - Generate every $\text{Seq. \#} \bmod k = 0$
 - Digest of state, last seq. #
 - Collect $2f + 1$ with seq. # and state digest
- Messages less than the last checkpoint # are dropped

Changing views as a backup

- Timeout waiting to execute request
- Multicast VIEW-CHANGE
 - stable checkpoint
 - proof: $2f+1$ checkpoint msgs.
 - pre-prepare for ops after checkpoint
 - proof: $2f$ matching prepare msgs.

Changing views - the new primary

- Get $2f$ VIEW-CHANGE for $v+1$
- Multicast NEW-VIEW
 - proof: $2f$ VIEW-CHANGE msgs.
 - include pre-prepare msgs.
 - between latest stable checkpoint and highest prepare msg.

Protecting liveness

- Operation fails - move to new view
- View change fails - move to new view
- Faulty nodes can not force view change

Safety

- Backups can not conflict for the same seq#
- Commits to different views are agreed upon

Optimizations

- One backup sends result, others send replica
- Replicas execute request tentatively
- Read only requests are sent immediately
 - Client still must wait for $2f + 1$ replies

Implementation of the Replication Library

- *invoke* - do something to the state machine
- UDP messages over IP and IP multicast
- Several API upcalls that the application must implement (execute, checkpoint, digest)
- Retransmissions are driven by the receiver

Implementation of BFS

- User level app performs BFT operations and mediates kernel NFS calls on client
- Replicas have user level program
 - Memory mapped file to store data on disk
 - Copy-on-write to save space when doing checkpoints
 - Record of changes post-checkpoint

Incremental state digest

- Need to maintain digest of state for checkpoints
 - $\text{digest} = (\text{Sum digests of blocks}) \bmod x$
- Incremental hash function
 - When updating a checkpoint
 - Add new hash to digest
 - Subtract old hash from digest

Overview of Evaluation

- Micro-Benchmark
 - Evaluate the **Replication Library**
- Andrew Benchmark
 - Evaluate **BFS**

Micro-Benchmark Results (worst case)

Latency to invoke null operation

arg/res (KB)	read-write	read-only	non repl
0/0	3.35 (309%)	1.62 (98%)	0.82
4/0	14.19 (207%)	6.98 (51%)	4.62
0/4	8.02 (72%)	5.94 (27%)	4.66
	replicated		

Replication adds **98%** overhead to no data null operation

Relative overhead **less significant** when moving data

Write requests add **significant** overhead

Andrew Benchmark

- BFS-strict takes only 26% more time to run than BFS-nr
- BFS-strict only 3% more time than NFS-std
- With read only lookup -- 2% faster!

The good and the bad

- Good:
 - The protocol is described in great detail
 - Appears that they have a BFT system
 - Optimizations are nice touch
- Bad:
 - Holy message flooding Batman!
 - The protocol is described in great detail
 - Implementation relies on OS mmap