Practical Byzantine Fault Tolerance

Robert Grimm
New York University

(Partially based on notes by Eric Brewer and David Mazières)

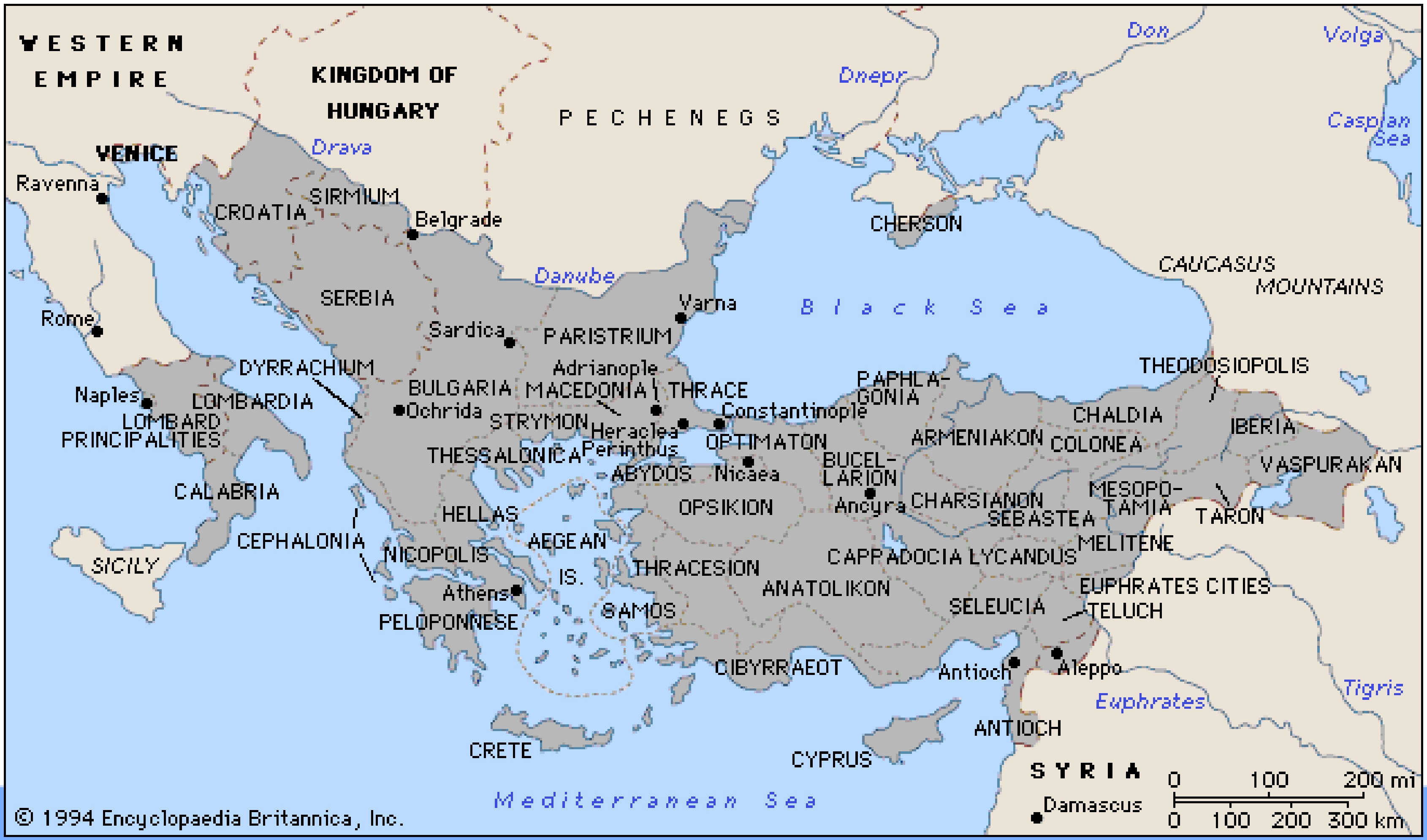
The Three Questions

- What is the problem?
- What is new or different?
- What are the contributions and limitations?

The Byzantine Empire

- "Eastern half of the Roman Empire, which survived for *a thousand years* after the western half had crumbled into various feudal kingdoms and which finally fell to Ottoman Turkish onslaughts in 1453"

[Encyclopaedia Britannica]



AD 1025

A Refined Culture…

- Byzantine |ˈbizənˌtēn; bəˈzan-; -ˌtīn|
    - of an ornate artistic and architectural style that developed in the Byzantine Empire and spread esp. to Italy and Russia

Angeli autẽ corpus eius accipientes ab illo loco
usq; ad montẽ Synay itinẽ plusq̃ dierum xx.
deduxerunt et ibidẽ sepelierũt ex cui° ossib;
idesinẽter oleũ manat infirmitates sanãs.

… Is Being Bad-Mouthed

- Byzantine |ˈbizənˌtēn; bəˈzan-; -ˌtīn|
  - of an ornate artistic and architectural style that developed in the Byzantine Empire and spread esp. to Italy and Russia
  - (of a system or situation) excessively complicated, typically involving a great deal of administrative detail
  - characterized by deviousness or underhanded procedure

[Mac OS Dictionary.app]

  - A system showing arbitrary, even malicious behavior

[Lamport et al., TOPLAS '82]

Background

(A)Synchronous Systems

- Synchronous: all nodes proceed in agreed-on "rounds"
  - Messages sent and received within rounds
- Asynchronous: steps occur at different times, speeds, and nodes
  - Obviously more realistic, but...
- Impossible to achieve consensus on (even) a single bit in an asynchronous system with one faulty node
  - Fundamentally, faults are indistinguishable from unbounded delays
  - [Fischer et al., JACM '83]

Linearizability

- Sequential consistency
  - "...the results of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program."
  - [Lamport, IEEE Transactions on Computers '79]
- Linearizability
  - Sequential consistency + respecting real-time order of events
  - [Herlihy & Wing, POPL '87]

Safety and Liveness

- Safety: bad things do not happen
  - BFT: does *not* depend on synchrony
- Liveness: good things happen eventually
  - BFT: depends on timing bounds, i.e., weak synchrony
    - delay(t) = o(t), i.e., delay does not grow faster than time

- Note that there are different definitions of "bad things"
  - Fail stop: faulty nodes just stop — is this realistic?
  - Byzantine: anything can happen
    - E.g., drop, delay, duplicate, or re-order messages

Core Algorithm

Goal and Basic Idea

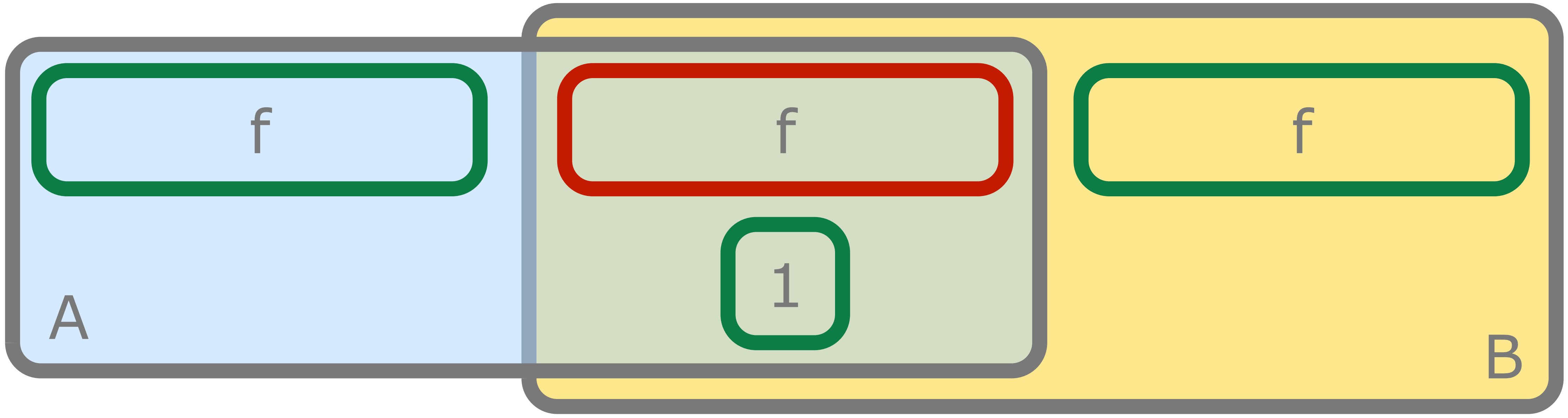
- Goal: build a linearizable replicated state machine
  - Replicate for fault prevention, not for scalability/availability
  - Agree on operations and their order
- Basic idea: get same statement from enough nodes to know that non-faulty nodes are in same state
  - Assume 3f+1 nodes, with at most f faults
  - Assume signed messages
  - Assume deterministic behavior
  - Assume no systematic failures

The cf+1 of BFT

- f+1 nodes
  - One node must be non-faulty, ensuring correct answer
- 2f+1 nodes
  - A majority of nodes must be non-faulty, providing *quorum*, i.e., locking in state of system
- 3f+1 nodes
  - A quorum must be available, even if f nodes are faulty

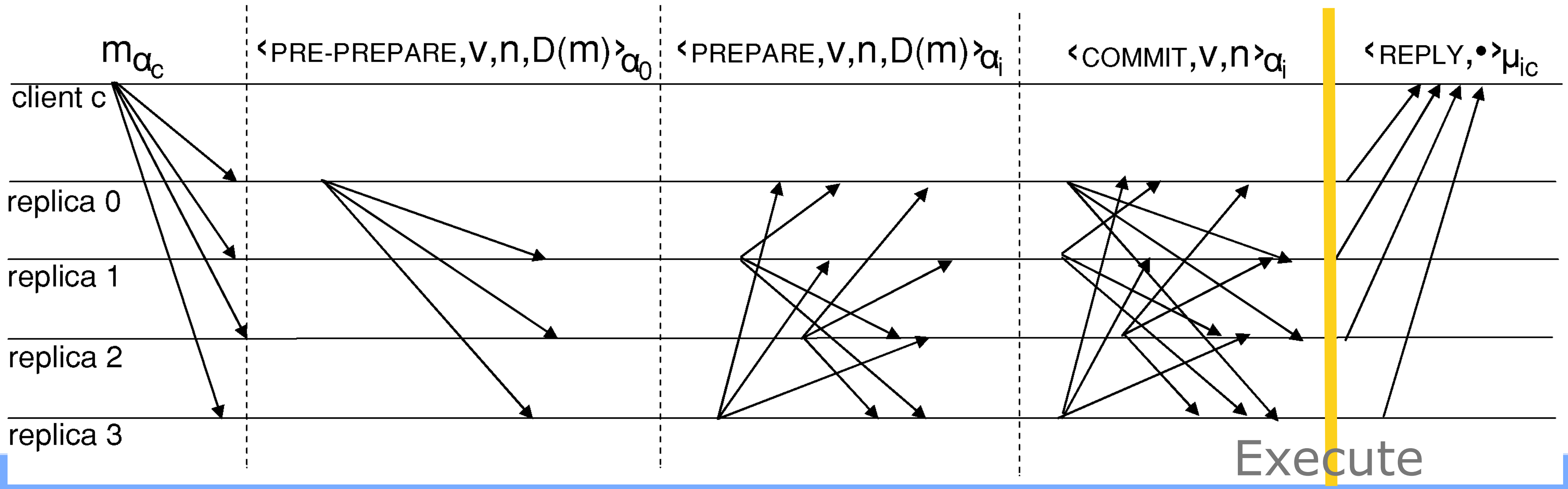
Properties of a Quorum

- Intersection
  - A gets 2f+1 responses with value x
  - B gets 2f+1 responses with value y
  - Then: x=y because A and B must share ≥1 non-faulty node
- Availability
  - With 3f+1 nodes, 2f+1 are non-faulty



Overview of Core Algorithm

- Client multicasts request to replicas
- Primary assigns order, backups agree to order, quorum
  - Three phases: pre-prepare, prepare, and commit
- Replicas execute requested operation, reply to client
- Client waits for f+1 responses with *same* value
  - At least one response comes from a non-faulty node


m_{α_c}
$\langle$PRE-PREPARE,v,n,D(m)$\rangle_{\alpha_0}$
$\langle$PREPARE,v,n,D(m)$\rangle_{\alpha_i}$
$\langle$COMMIT,v,n$\rangle_{\alpha_i}$
$\langle$REPLY,•$\rangle_{\mu_{ic}}$
client c
replica 0
replica 1
replica 2
replica 3
Execute

A View From a Client

- Client multicasts request $<REQUEST, o, t, c>_c$
  - o = operation, t = local timestamp, c = client id
- Client collects replies $<REPLY, v, t, c, i, r>_i$
  - v = view number, i = replica number, r = result
- Client uses result from f+1 correct replies
  - Valid signatures
  - f+1 different values of i
  - But same t and r

There's Work to Do

- Replica accepts <REQUEST, o, t, c>$_c$
  - Ensures signature is valid
  - Logs request to track progress through algorithm

Pre-Prepare Phase

- Primary multicasts <PRE-PREPARE, v, n, d>$_{\mathbf{p}}$
  - n = sequence number, d = digest of request
  - Proposes commit order for operation
- Backups accept and log pre-prepare message
  - Signature is valid, d matches actual request
  - Backup is in view v
  - Backup has *not* accepted different message for n
- Backups enter prepare phase

Prepare Phase

- Backups multicast <PREPARE, v, n, d, i>$_i$ to all replicas
- Replicas accept and log prepare messages
    - If signatures, v, n, and d match
- Operation is prepared on replica #i iff
    - #i has pre-prepare msg and 2f matching prepare msgs
- Once prepared, replica does ... ?

Are We There Yet? No!

- Prepared certificate says “we agree on n for m in v”
  - Still need to agree on order across view changes
- Alternative take
  - If prepared, replica #i knows there is a quorum
    - But messages can be lost etc., so others may not know
    - So, we still need to agree that we have quorum
- Solution: one more phase
  - Once prepared, replica enters commit phase

Commit Phase

- All replicas multicast <COMMIT, v, n, i>$_i$
  - Also accept and log others' correct commit messages
- Operation is committed on replica #i iff
  - Operation is prepared
  - Replica #i has 2f+1 matching commit messages (incl. own)
- Once committed, replica is ready to perform operation
  - But only after all operations with lower sequence numbers have been performed

The Log

- So far: log grows indefinitely
    - Clearly impractical
- Now: periodically checkpoint the state
    - Each replica computes digest of state
    - Each replica multicasts digest across replicas
    - 2f+1 such digests represent quorum (lock-in)
        - Can throw away log entries for older state, which is captured in *stable* checkpoint

View Change Algorithm

Primary May Be Faulty

- Goal: change view if no progress
  - View change is performed only for liveness
  - Primary selection is deterministic: primary = v mod |R|
    - Faulty nodes can be primary only for f views
  - Views overlap
    - Re-establish on-going operations by re-issuing pre-prepare and prepare messages
- Two algorithms
  - Based on public key cryptography [OSDI '99]
  - Based on replicas validating statements [OSDI '00, TOCS '02]

Let's Change the View

- If primary appears faulty, stop with core algorithm
- Backup multicasts <VIEW-CHANGE, v+1, n, C, P, i>$_i$
  - n = sequence # of last stable checkpoint s
  - C = set of 2f+1 checkpoint messages, i.e., proof of s
  - P = set of sets P_m for each m prepared at i with seq# > n
    - P_m = 1 pre-prepare msg + 2f prepare messages

We Have a New View

- Primary for v+1 collects 2f valid view-change msgs
  - As well as his own
- Primary multicasts <NEW-VIEW, v+1, V, O>$_{\mathbf{p}}$
  - V = set of view-change messages
  - O = set of pre-prepare messages
    - After last stable checkpoint
    - Contained in at least *one* P of view-change messages
      - Use no-op for sequence number gaps

Some Contingencies

- Replica #i does not have checkpoint
  - Retrieve from other replica
- O is incorrect, e.g., contains no-op for prepared op
  - Validate O locally and reject incorrect new-view msg
- We don't want to rely on public key cryptography
  - Study TOCS '02 paper in detail...

BFT in Practice

"Just a Library"

- Substantial piece of code
    - 8,500 lines of header files
    - 13,000 lines of C++ code
    - Leverages already substantial SFS library
- Simple interface
    - Client: *init* and *invoke*
    - Server: *init*, *execute*, and *modify*
        - *nondet* to agree on nondeterministic inputs
- Real application: BFT version of NSF v2 called BFS
    - 800 lines of header files, 2,500 lines of C code

Several Optimizations

- Use symmetric cryptography (where possible)
  - Use private keys only to establish session keys
- Send digest replies for large data items
  - Only need one copy
- Execute requests without waiting for commit quorum
  - May be aborted on view change
- Execute read-only requests even more eagerly
  - May fail in presence of concurrent writes
- Represent state as partition tree
  - Recursively compute digest
- Maintain checkpoints through copy-on-write

BFS vs. NFS

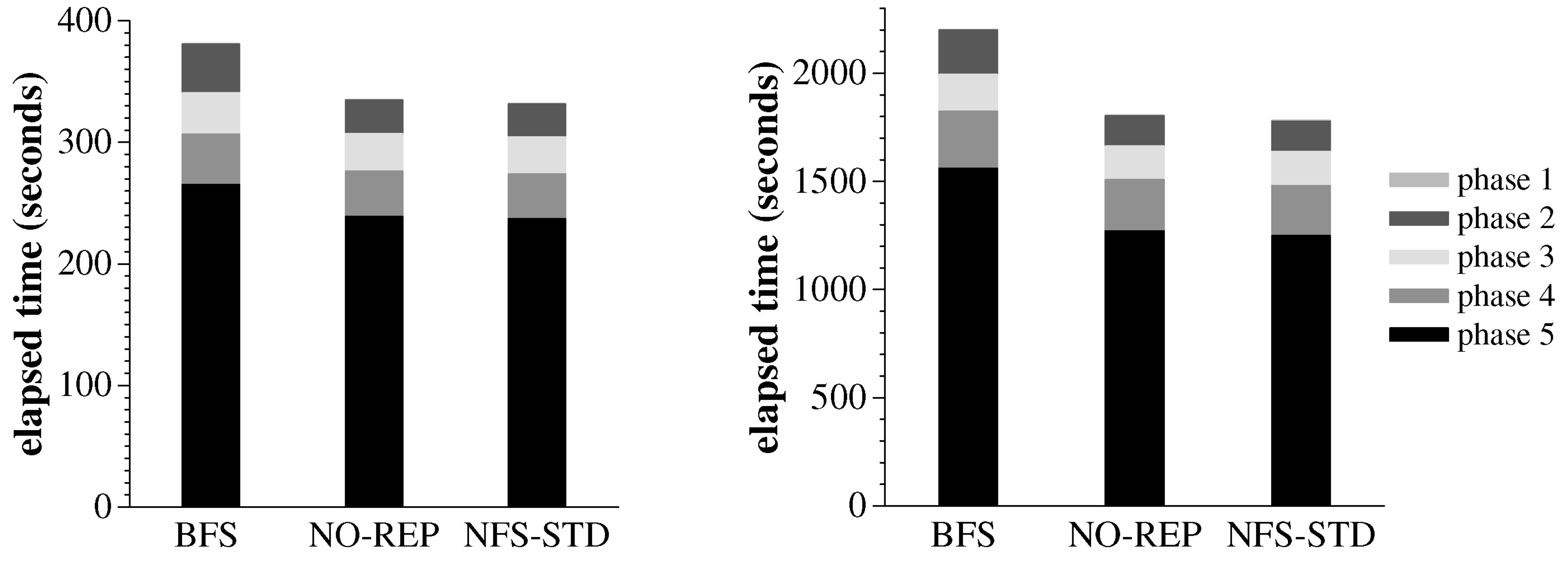


Fig. 15. Andrew100 and Andrew500: elapsed time in seconds.

- Andrew benchmark up to 22% slower

BFS vs. NFS (cont.)

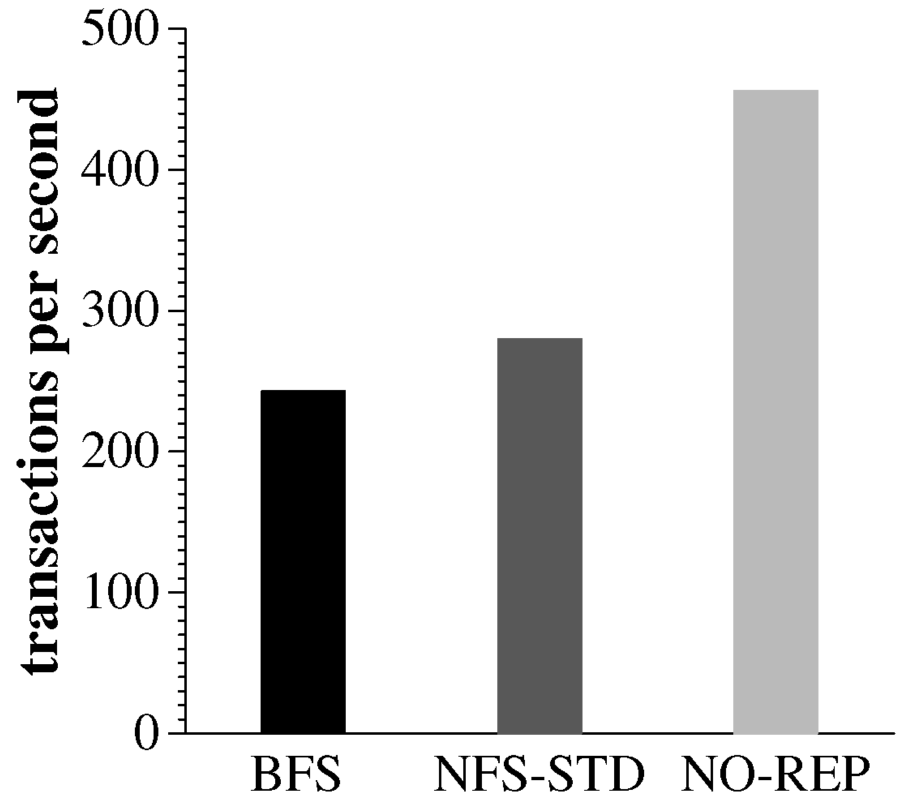


Fig. 16. PostMark: throughput in transactions per second.

- PostMark benchmark only 13% less throughput

Proactive Recovery

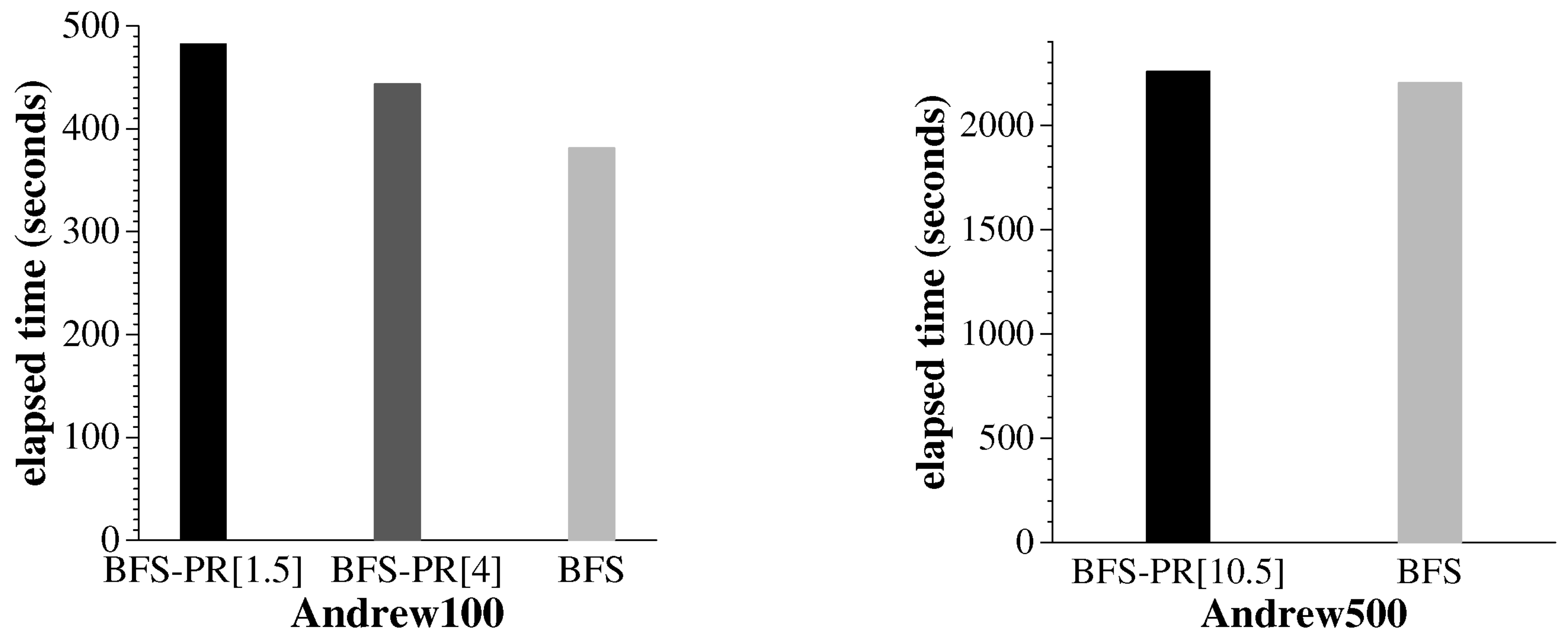


Fig. 17. Andrew: elapsed time in seconds with and without proactive recoveries.

- Adds up to another 27% over basic BFT
  - So, why do it?

What Do You Think?