Practical Byzantine Fault Tolerance

Miguel Castro and Barbara Liskov

MIT Laboratory for Computer Science

Why Byzantine Fault Tolerance?

- Traditional fault tolerance:
  - Processes fail by stopping or omitting steps
- Byzantine fault tolerance:
  - "No" assumptions on faulty behavior
  - Robust to increasingly common faults:
    - Hacker-tolerance
    - Bug-tolerance

Previous Work

- Mostly theoretical
  - Few implementations
  - Little analysis
- Rely on synchrony for correctness
  - Attack: delay nodes or communication
- Slow

[Rampart,SecureRing,Phalanx,…]

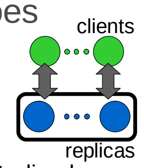
Contributions

- Practical:
  - Correct in asynchronous systems
  - Liveness under attack
  - Fast
- Implementation
  - Generic replication library
  - BFS – a Byzantine-fault-tolerant NFS
- Performance evaluation

Talk Overview

- Algorithm
- Optimizations
- BFS
- Performance evaluation
- Conclusions

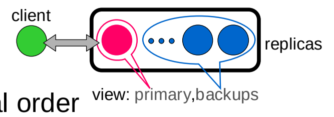
What the Algorithm Does



- Arbitrary replicated service
- Safety and liveness:
  - Service behaves as a correct centralized one
  - Clients eventually receive replies to requests
- Assumptions:
  - 3f+1 replicas to tolerate f faults (optimal)
  - Strong cryptography (reasonable)
  - Unknown eventual bounds (only for liveness)

Algorithm Overview

State machine replication
- Deterministic replicas start in same state
- Execute same requests in same order
- Client waits for f+1 matching replies



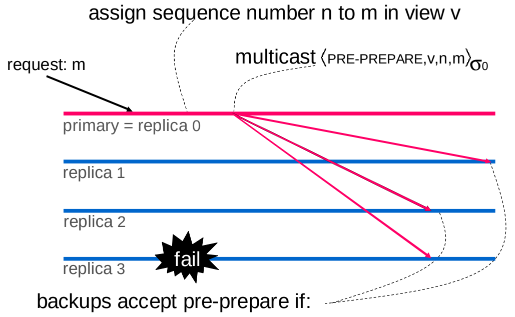
To agree on a total order
- Primary picks ordering
- Backups ensure primary behaves
  - certify correct actions
  - trigger view changes

Ensuring Safety

- Three phase protocol:
  - pre-prepare, prepare and commit
  - pre-prepare and prepare order within views
  - prepare and commit order across views
- Messages are authenticated
  - $\langle \bullet \rangle_{\sigma_I}$ denotes a messaged signed by I
- Replicas remember messages received in log

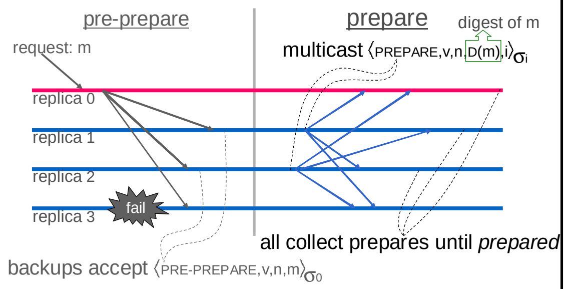
Normal Case: Pre-prepare Phase

assign sequence number n to m in view v



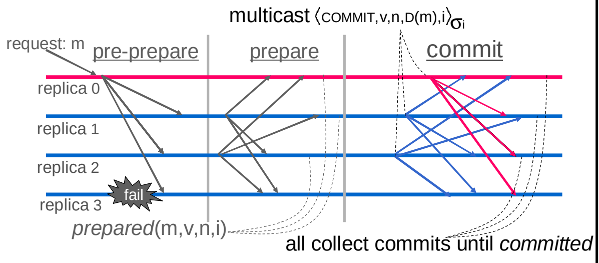
backups accept pre-prepare if:
- in view v
- never accepted $\langle$PRE-PREPARE,v,n, m'$\rangle_{\sigma_0}$ with m' $\neq$ m

Normal Case: Prepare Phase


pre-prepare
prepare
digest of m
request: m
multicast $\langle$PREPARE,v,n,D(m),i$\rangle_{\sigma_i}$
replica 0
replica 1
replica 2
replica 3
fail
all collect prepares until *prepared*
backups accept $\langle$PRE-PREPARE,v,n,m$\rangle_{\sigma_0}$

- *prepared*(m,v,n,i) $\equiv$ pre-prepare for m,v,n + 2f matching prepares
- Order within view :
  - Distinct m and m' are never *prepared* for same v and n

Normal Case: Commit Phase


multicast $\langle$COMMIT,v,n,D(m),i$\rangle_{\sigma_i}$
request: m
pre-prepare
prepare
commit
replica 0
replica 1
replica 2
replica 3
fail
prepared(m,v,n,i)
all collect commits until *committed*

- *committed*(m,v,n,i) $\equiv$ *prepared* and 2f+1 commits for m,v,n
- Execute after all m' with lower sequence numbers
- If *committed*(m,v,n,i), *prepared*(m,v,n,i) for f+1 non-faulty

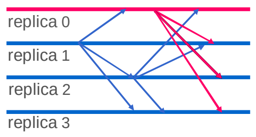
View Changes

- Liveness when primary fails:
  - Backups multicast view-change messages
  - Primary $\equiv$ view number modulo number of replicas
  - New primary multicasts new-view message
- Ordering across views:
  - Information about *prepared* requests in view-changes
  - New-view message:
    - includes 2f+1 view-change messages
    - contains *committed* request information
    - only accept messages consistent with new-view

Distinct m and m' never *committed* for same n

Garbage Collection

- Discard logged information after having proof:
  – request was executed by f+1 non-faulty
  – state after request execution is correct



- periodically checkpoint state
- multicast $\langle \text{CHECKPOINT},v,n,\boxed{D(state)},i \rangle_{\sigma_i}$

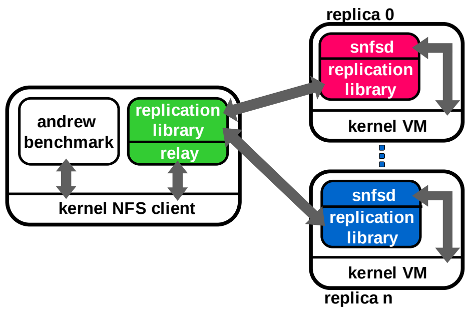
digest of checkpoint

- Proof = 2f+1 matching checkpoint messages
- Discard messages and checkpoints that precede proof
- Efficient: copy-on-write and incremental digest of checkpoints

Optimizations

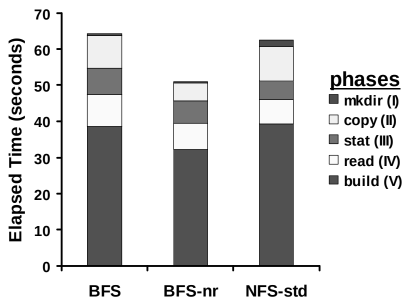
- Digest replies: only one reply with full result
- Optimistic execution: execute *prepared* requests
  – Operations execute in 2 round-trips
- Read-only operations: executed in current state
  – Read-only operations execute in 1 round-trip
- Fast authentication: MACs in normal case
  – MAC 1000x faster than public-key signatures
  – Non-trivial: cannot prove authenticity to third party

BFS - A Byzantine-Fault-Tolerant NFS



No synchronous writes – stability through replication

Andrew Benchmark



- BFS-nr is like BFS but without replication
- NFS-std is the Digital Unix NFS V2 implementation

Conclusions

Byzantine fault tolerance is practical:
- Low impact on latency
- Works in asynchronous systems

Extensions:
- Recovery
- Fault-tolerant privacy
- Witnesses
- Reduce number of copies of state