# Byzantine Agreement

**Yee Jiun Song**

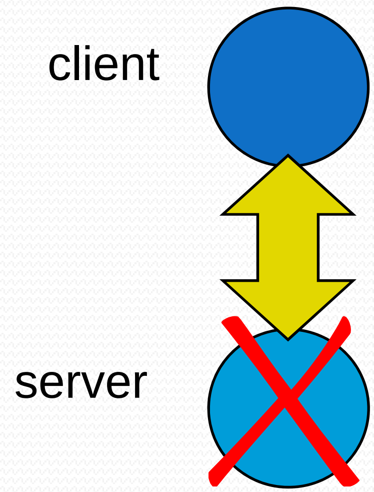
***Cornell University. CS5410 Fall 2008.***

# Fault Tolerant Systems

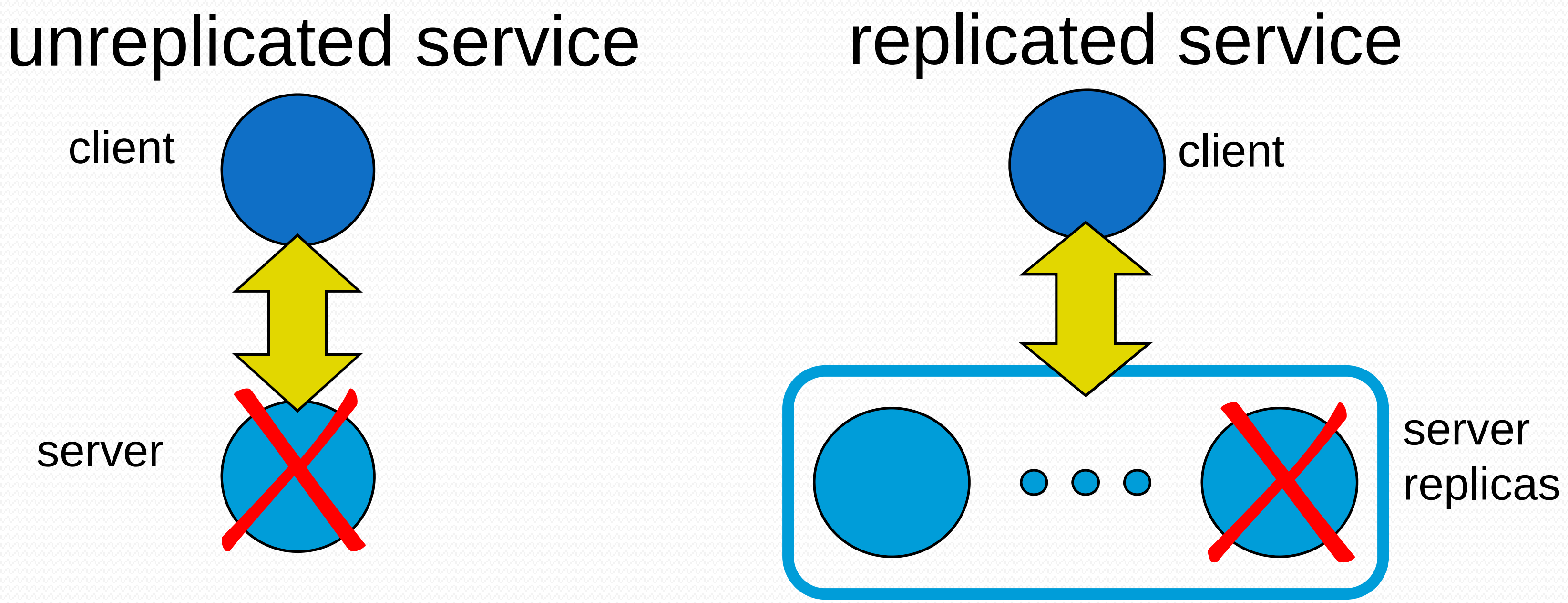
- By now, probably obvious that systems reliability/availability is a key concern
- Downtime is expensive
- Replication is a general technique for providing fault tolerance

# Replication

unreplicated service



# Replication



# Replication

- Applications as deterministic state machines
- Reduce the problem of replication to that of *agreement*
- Ensure that replicas process requests in the same order:
  - Safety: clients never observe inconsistent behavior
  - Liveness: system is always able to make progress

# Traditional Assumptions

- Synchrony
  - Bounded difference in CPU speeds
  - Bounded time for message delivery
- Benign/Crash faults
  - When machines fail, they stop producing output immediately, and forever.

What if these assumptions don’t hold?

# Asynchrony

- In the real world, systems are never quite as synchronous as we would like
- Asynchrony is a pessimistic assumption to capture real world phenomenon
  - Messages will eventually be delivered, processors will eventually complete computation. But no bound on time.
- In general:
  - OK to assume synchrony when providing liveness
  - Dangerous (NOT OK) to assume synchrony for safety

# Byzantine Faults

- Crash faults are a strong assumption
- In practice, many kinds of problems can manifest:
  - Bit flip in memory
  - Intermittent network errors
  - Malicious attacks
- Byzantine faults: strongest failure model
  - Completely arbitrary behavior of faulty nodes

# Byzantine Agreement

- Can we build systems that tolerate Byzantine failures and asynchrony? YES!
- Use replication + Byzantine agreement protocol to order requests
- Cost
  - At least 3t+1 replicas (5t+1 for some protocols)
  - Communication overhead
- Safety in the face of Byzantine faults and asynchrony
- Liveness in periods of synchrony

# PBFT

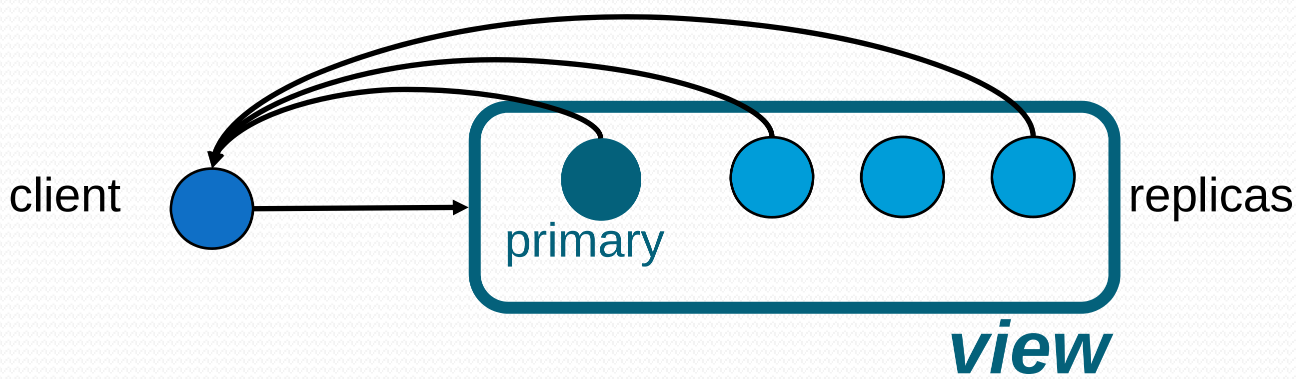
- Castro and Liskov. "Practical Byzantine Fault Tolerance." OSDI99.
- The first replication algorithm that integrates Byzantine agreement
- Demonstrates that Byzantine Fault-Tolerance is not prohibitively expensive
- Sparked off a thread of research that led to the development of many Byzantine fault-tolerant algorithms and systems

# PBFT: Overview

- Servers are replicated on 3t+1 nodes
- One particular server is called the *primary*. Also called the *leader* or the *coordinator*
- A continuous period of time during which a server stays as the *primary* is called a *view*, or a *configuration*

# PBFT: Normal Operation

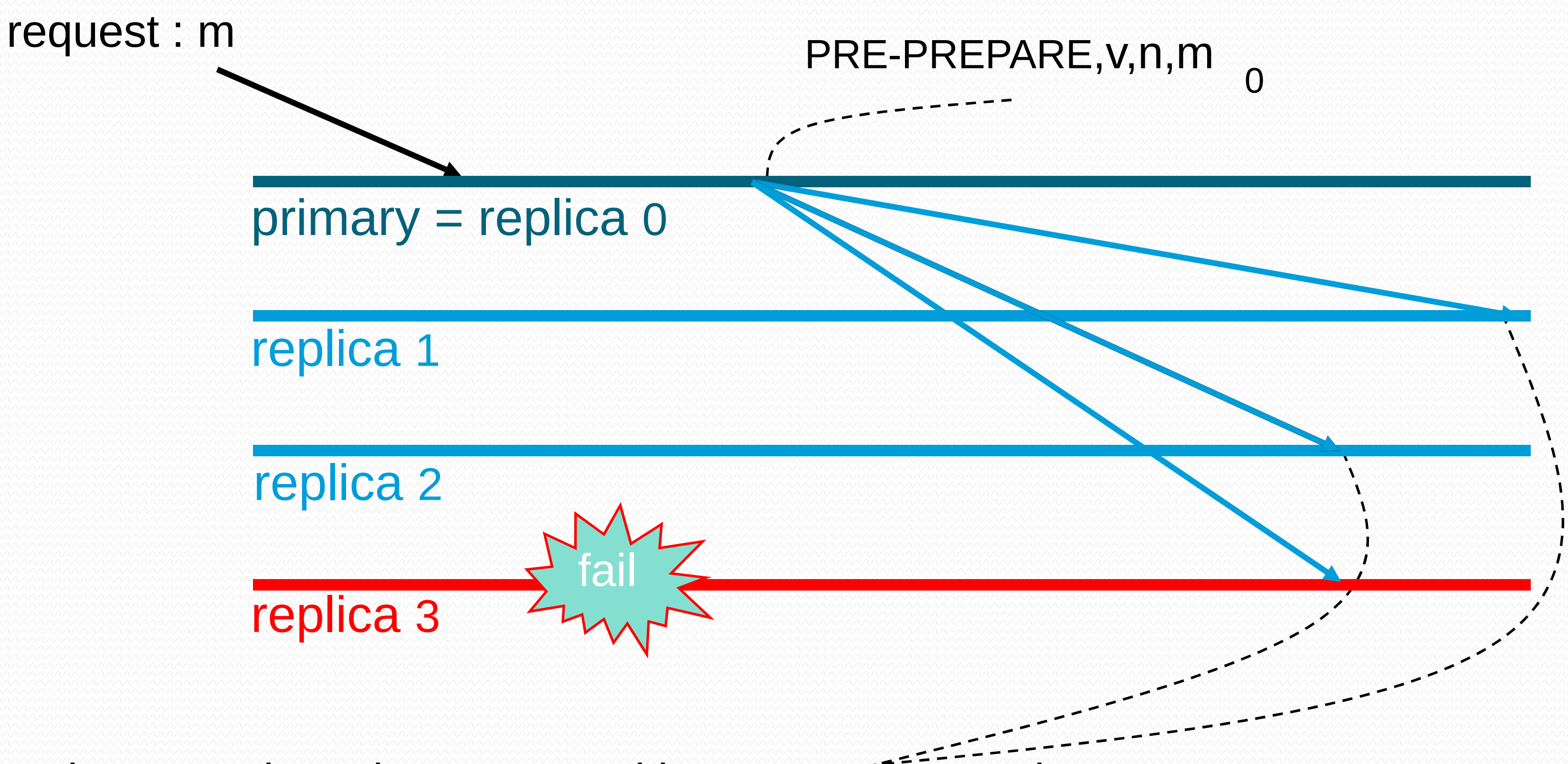
- Fixed primary within a view
- Client submits request to primary
- Primary orders requests and sends them to all nodes
- Client waits for identical replies from at least t+1 nodes



# Client

- Waits for t+1 identical replies
- Why is this sufficient?
  - At most t failures. So at least one of the (t+1) replies must be from a correct node.
  - PBFT ensures that non-faulty nodes never go into a bad state, so their responses are always valid.
  - Difficult: How to ensure this is the case?
- If client times out before receiving sufficient replies, broadcast request to all replicas

# Phase 1: Pre-prepare

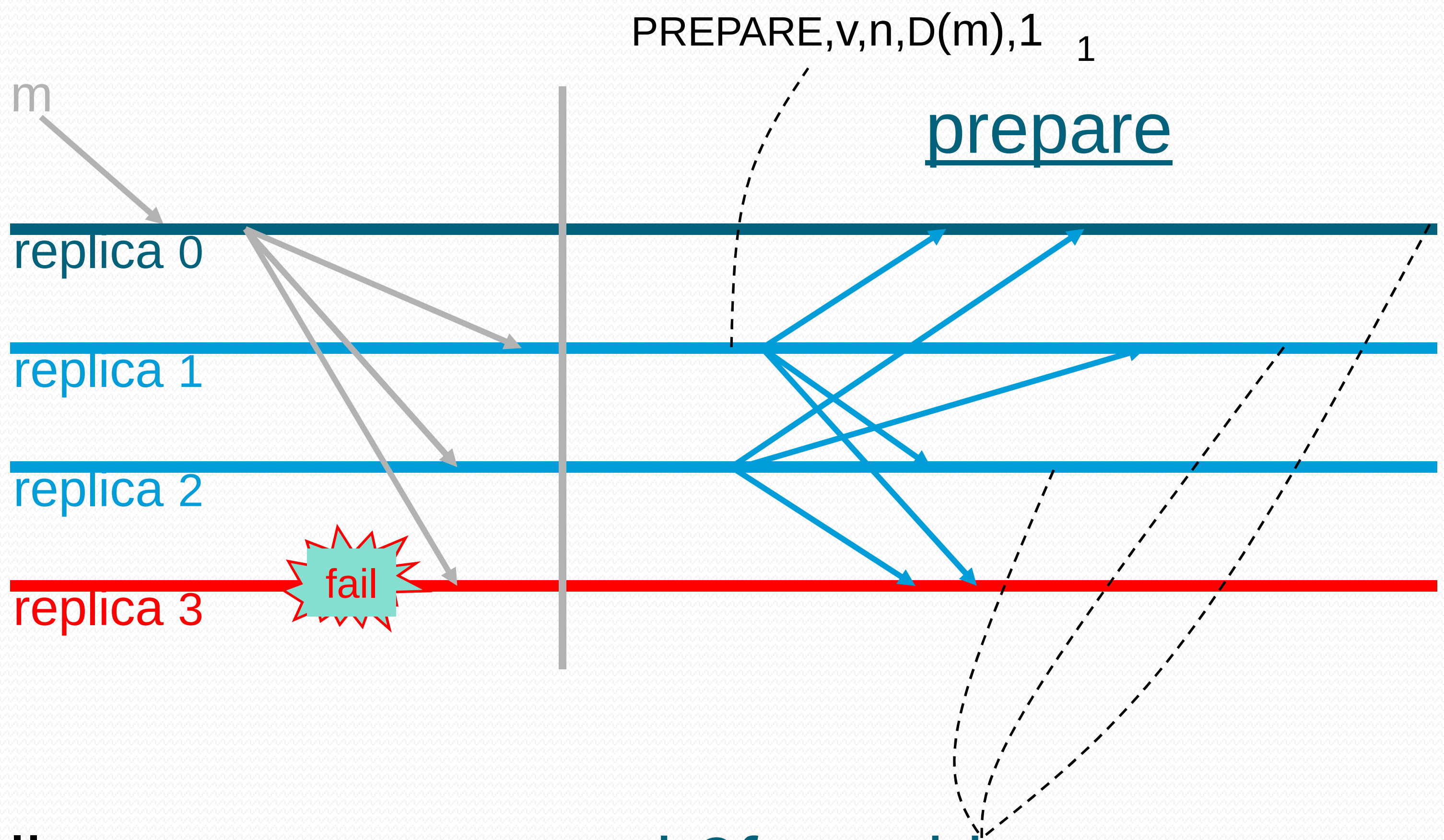



Primary assigns the request with a sequence number n

Replicas accept pre-prepare if:

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

# Phase 2: Prepare



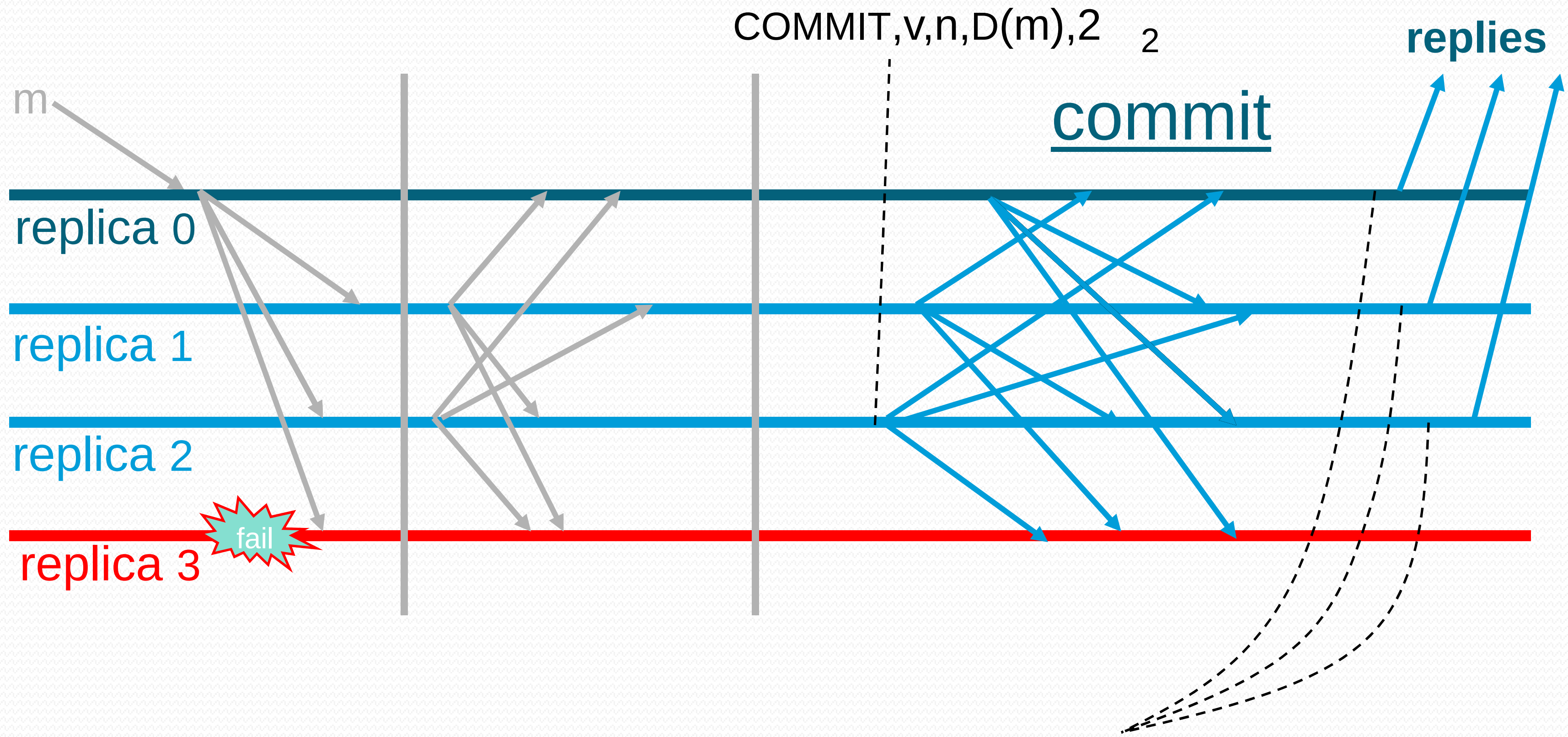

collect pre-prepare and *2f* matching prepares

***P-certificate(m,v,n)***

# Phase 2: Prepare

- Each replica collects 2f prepare msgs:
  - 2f msgs means that 2f+1 replicas saw the same pre-prepare msg. At least f+1 of these must be honest
  - Since there are only 3f+1 replicas, this means that there cannot exist more than 2f replicas that received a conflicting pre-prepare msg or claim to have received one
  - All correct replicas that receive 2f prepare msgs for a <v, n, m> tuple received consistent msgs

# Phase 3: Commit




all collect 2f+1 matching commits

***C-certificate(m,v,n)***

Request m executed after:

- having *C-certificate(m,v,n)*
- executing requests with sequence number less than n

# Phase 3: Commit

- If a correct replica p receives 2f+1 matching commit msgs
  - At least f+1 correct replicas sent matching msgs
  - No correct replica can receive 2f+1 matching commit msgs that contradict with the ones that p saw
- In addition, phase 2 ensures that correct replicas send the same commit msgs, so, together with the view change protocol, correct replicas will eventually commit

# Why does this work?

- When a replica has collected sufficient *prepared* msgs, it knows that sufficient msgs cannot be collected for any other request with that sequence number, in that view
- When a replica collects sufficient *commit* msgs, it knows that eventually at least *f+1* non-faulty replicas will also do the same
- Formal proof of correctness is somewhat involved. Refer to paper. Drop by my office (320 Upson) if you need help.

# View Change

- What if the primary fails? View change!
- Provides liveness when the primary fails
- New primary = view number mod N
- Triggered by timeouts. Recall that the client broadcasts the request to all replicas if it doesn't receive sufficient consistent requests after some amount of time. This triggers a timer in the replicas.

# View Change

- A node starts a timer if it receives a request that it has not executed. If the timer expires, it starts a view change protocol.
- Each node that hits the timeout broadcasts a VIEW-CHANGE msg, containing certificates for the current state
- New primary collects 2f+1 VIEWCHANGE msgs, computes the current state of the system, and sends a NEWVIEW msg
- Replicas check the NEWVIEW msg and move into the new view

# PBFT Guarantees

- Safety: all non-faulty replicas agree on sequence numbers of requests, as long as there are <= t Byzantine failures
- Liveness: PBFT is dependent on view changes to provide liveness. However, in the presence of asynchrony, the system may be in a state of perpetual view change. In order to make progress, the system must be synchronous enough that some requests are executed before a view change.

# Performance Penalty

- Relative to an unreplicated system, PBFT incurs 3 rounds of communication (pre-prepare, prepare, commit)
- Relative to a system that tolerates only crash faults, PBFT requires 3t+1 rather than 2t+1 replicas
- Whether these costs are tolerable are highly application specific

# Beyond PBFT

- Fast Byzantine Paxos (Martin and Alvisi)
  - Reduce 3 phase commit down to 2 phases
  - Remove use of digital signatures in the common case
- Quorum-based algorithms. E.g. Q/U (Abu-El-Malek et al)
  - Require 5t+1 replicas
  - Does not use agreement protocols. Weaker guarantees. Better performance when contention is low.

# Zyzzyva (Kotla et al)

- Use speculation to reduce cost of Byzantine fault tolerance
- Idea: leverage clients to avoid explicit agreement
  - Sufficient: Client knows that the system is consistent
  - Not required: Replicas know that they are consistent
- How: clients commits output only if they know that the system is consistent

# Zyzzyva

- 3t+1 replicas
- As in PBFT, execution is organized as a sequence of views
- In each view, one replica is designated as the primary
- Client sends request to the primary, the primary forwards the request to replicas, and the replicas execute the request and send responses back to clients

# Zyzzyva

- If client receives 3t+1 consistent replies, it’s done
- If client receives between 2t+1 and 3t consistent replies, the client gathers 2t+1 responses and distributes a “commit certificate” to the replicas. When 2t+1 replicas acknowledge receipt of the certificate, the client is done.

# Zyzzyva: Caveats

- Correct replicas can have divergent state. Must have a way to reconcile differences.
- View change protocol significantly more complicated, since replicas may not be aware of a committed request (only a client knew, by receiving 3t+1 identical replies)
- Performance is timeout sensitive. How long do clients wait to see if they'll receive 3t+1 identical replies?

# Beyond Zyzzyva

- In the good case, Zyzzyva takes 3 network latencies to complete (Client→Primary→Replicas→Client). Is is possible to eliminate yet another round of communication to make Byzantine Fault Tolerance perform as well as an unreplicated system?
- Yes! If clients broadcast requests directly to all replicas, leaderless protocols are available that can allow requests to complete in 2 network latencies (Client→Replicas→Client).

# Bosco: Byzantine One-Step Consensus

- In the absence of contention, Byzantine agreement is possible in one communication step
- Strong one-step Byzantine agreement:
  - One-step performance even in the presence of failures
  - 7t+1 replicas
- Weak one-step Byzantine agreement:
  - One-step performance only in the absence of failures and contention
  - 5t+1 replicas

# Practical Concerns

- State machine replication is a popular approach to provide fault tolerance in real systems
  - Chubby (Google) and Zookeeper (Yahoo) are toolkits that are essentially built on top of agreement protocols
- But *Byzantine* fault tolerant systems are not as common – why?
  - Application specific checks can be used to mask/detech non-crash faults.
  - Performance overhead significant
    - More machines
    - More network overhead

# Practical Concerns

- As machines/bandwidth become cheaper, and downtime become more intolerable – will this change?
- Can BFT help make applications easier to write?
- Can a combination of BFT, code obfuscation, and other techniques make systems more secure?

# References


[1] Miguel Castro and Barbara Liskov. Practical Byzantine Fault Tolerance. OSDI 1999.

[2] Michael Abd-El-Malek, Gregory R. Granger, Garth R. Goodson, Michael K. Reiter, Jay J. Wylie. Fault-Scalable Byzantine Fault-Tolerant Services. SOSP 2005.

[3] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, Edmund Wong. Zyzzyva: Speculative Byzantine Fault Tolerance. SOSP 2007.

[4] Jean-Philippe Martin and Lorenzo Alvisi. Fast Byzantine Consensus. IEEE TODSC 2006.

[5] Yee Jiun Song and Robbert van Renesse. Bosco: One-Step Byzantine Asynchronous Consensus. DISC 2008.

# Happy Thanksgiving!