Byzantine Fault Tolerance

(h/t Ellis Michael and Dan Ports)

Failure models

- Fail-stop: nodes either execute the protocol correctly or just stop
- Byzantine failures: nodes can behave in any arbitrary way
  - Send illegal messages, try to trick other nodes, collude, …
- Why this model?
  - Consequences of software bugs are often unpredictable; measurable rate of (not fail stop) hardware failures
  - Build systems that don't rely on everyone being trusted

What can go wrong?

A: Append(x, "foo"); Append(x, "bar")
B: Get(x) -> "foo bar"
C: Get(x) -> "foo bar"

- What can a malicious server do?
  - return something totally unrelated
  - reorder the append operations ("bar foo")
  - only process one of the appends
  - show B and C different results

Paxos is fail-stop tolerant

- Paxos tolerates up to f out of 2f+1 ***fail-stop*** failures
- What could a malicious replica do?
  - stop processing requests (but Paxos should handle this!)
  - change the value of a key
  - acknowledge an operation then discard it
  - execute and log a different operation
  - tell some replicas that slot 42 is Put and others that it is Get
  - force view changes to keep the system from making progress

BFT replication

- Same replicated state machine model as Paxos
- Tolerate f byzantine failures out of 3f+1 replicas
- Other 2f+1 replicas are non-faulty, but might be slow
- Use voting, signatures so that the correct replicas return the right result
- If client hears the same thing from f+1 replicas, done!

BFT model

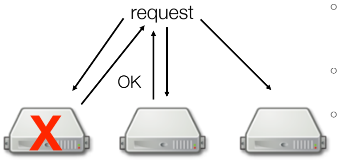
- Attacker controls f replicas
  - can make them do anything
  - knows their crypto keys, can send messages
- Attacker knows what protocol the other replicas are running
- Attacker can delay messages in the network arbitrarily
- But the attacker can't
  - cause more than f replicas to fail
  - cause clients to misbehave
  - break crypto

Why is BFT consensus hard?

...and why do we need 3f+1 replicas?

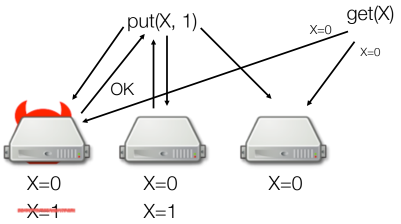
Paxos Quorums

- Why did Paxos need 2f+1 replicas to tolerate f failures?
- Every operation needs to talk w/ a majority (f+1)
  - f of those nodes might fail
  - need one left
  - quorums intersect



The Byzantine case

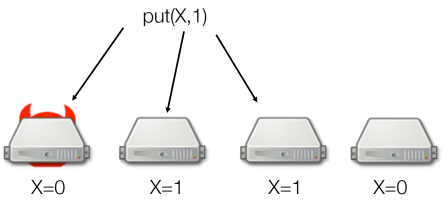
- What if we tried to tolerate Byzantine failures with 2f+1 replicas?



Quorums

- In Paxos: quorums of f+1 out of 2f+1 nodes
  - quorum intersection:
    any two quorums intersect at at least one node
- For BFT: quorums of 2f+1 out of 3f+1 nodes
  - quorum ***majority***
    any two quorums intersect at *f+1* nodes
    => any two quorums intersect at at least one good node

Are quorums enough?



Are quorums enough?

- We saw this problem before with Paxos:
  just writing to a quorum wasn't enough
- Solution in Paxos:
  - use a two-phase protocol: propose, then accept

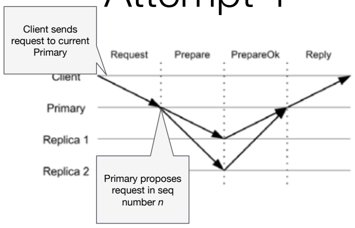
BFT approach

- Use a primary to order requests
- But the primary might be faulty
  - ○ could send wrong result to client
  - ○ could ignore client request entirely
  - ○ could send different op to different replicas (this is the really hard case!)

BFT approach

- All replicas send replies directly to client
- Replicas exchange information about ops received from primary (to make sure the primary isn't equivocating)
- Clients notify all replicas of ops, not just primary; if no progress, they replace primary
- All messages cryptographically signed; serve as transferrable proof (e.g., I know you received message X)

Attempt 1



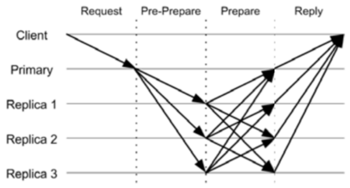
Client sends request to current Primary

Primary proposes request in seq number *n*

- What's the problem with using this?
  - primary might send different op order to replicas

Attempt 2

- Client sends request to primary & other replicas
- Primary assigns seq number, sends PRE-PREPARE(seq, op) to all replicas
- When replica receives PRE-PREPARE, sends PREPARE(seq, op) to all replicas
  - ○ 2f+1 PREPAREs serve as proof certificate, to anyone
  - ○ Once it has proof, the replica executes and replies to the client
  - ○ Client can proceed when it hears f+1 (same) replies



- Can a faulty non-primary replica prevent progress?
- Can a faulty primary cause a problem that won't be detected?
  - ○ What if it sends ops in a different order to different replicas?

Faulty primary

- What if the primary sends different ops to different replicas?
  - ○ case 1: all good nodes get 2f+1 matching prepares
    - ▪ they must have gotten the same op
  - ○ case 2: >= f+1 good nodes get 2f+1 matching prepares
    - ▪ they must have gotten the same op
    - ▪ what about the other (f or less) good nodes?
  - ○ case 3: < f+1 good nodes get 2f+1 matching prepares
    - ▪ system is stuck, doesn't execute any request

View changes

- What if a replica suspects the primary of being faulty?
  e.g., heard request but not PRE-PREPARE
- Can it start a view change on its own?
  - no - it needs f+1 view change
- Who will be the next primary?
  - How do we keep a malicious node from making sure it's always the next primary?
  - primary = view number mod n

Straw-man view change

- When a replica suspects the primary, sends VIEW-CHANGE to the next primary, includes all of the PREPARE certificates it received. Asks other replicas to join in.
- Other replicas join the view change when they receive f+1 requests.
- Once primary receives 2f+1 VIEW-CHANGEs,
  announces view with NEW-VIEW message
  - includes copies of the VIEW-CHANGES (showing the view change is justified; propagates the PREPARE certificates)
  - starts numbering new operations after last seq number it saw

What goes wrong?

- Some replica saw 2f+1 PREPAREs for an op in seq number *n*, executed it
- The new primary did not receive the PREPARE for that op
- New primary starts numbering new requests at n
  => two different ops with seq num n!

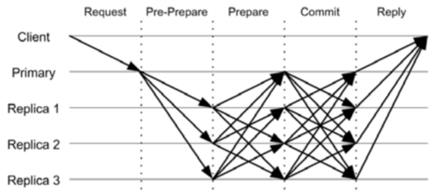
Fixing view changes

- Need another round in the operation protocol!
- Not just enough to know that replicas agreed on an op for seq *n*, need to make sure that the next primary will hear about it
- After receiving 2f+1 PREPAREs, replicas send COMMIT message to let the others know
- Only execute requests after receiving 2f+1 COMMITs;
  receiving 2f+1 COMMITs is a certificate that **any quorum** contains f+1 nodes with the PREPARE certificate

The final protocol

- client sends op to primary
- primary sends PRE-PREPARE(seq, op) to all
- all send PREPARE(seq, op) to all
- after replica receives 2f+1 matching PREPARE(seq, op), send COMMIT(seq, op) to all
- after receiving 2f+1 matching COMMIT(seq, op), execute op, reply to client

The final protocol



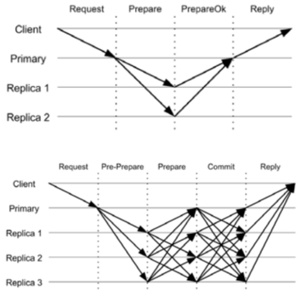
The final protocol

- Correct clients only accept replies from f+1 replicas
- Correct replicas only execute once they have a COMMIT certificate, implying that f+1 correct replicas have a PREPARE certificate
- Therefore, if a replica has a COMMIT certificate, that operation will survive in that seq into new views
- Replicas never send conflicting PREPAREs in the same view
- Therefore, no two correct replicas ever execute different operations for the same seq number

BFT vs MultiPaxos

- BFT: 4 phases
  - PRE-PREPARE - primary determines request order
  - PREPARE - replicas make sure primary told them same order
  - COMMIT - replicas ensure that a quorum knows about the order
  - execute and reply
- MultiPaxos: 3 phases
  - PREPARE - primary determines request order
  - PREPARE-OK - replicas ensure that a quorum knows about the order
  - execute and reply

PBFT vs MultiPaxos



What did this buy us?

- Before, we could only tolerate fail-stop failures with replication
- Now we can tolerate ***any*** failure, benign or malicious
  - as long as it only affects less than 1/3 replicas
  - (what if more than 1/3 replicas are faulty?)

Performance

- Why would we expect BFT to be slow?
  - Latency (extra round)
  - Message complexity (O(n^2) communication!)
  - Crypto ops are slow!

Implementation Complexity

- Building a bug-free Paxos is hard!
- BFT is much more complicated
- Which is more likely?
  - bugs caused by the BFT implementation
  - the bugs that BFT is meant to avoid

BFT summary

- It’s possible to build systems that work correctly even though parts may be malicious!
- Requires a lot of complex and expensive mechanisms
- On the boundary of practicality?