Byzantine Fault Tolerance

Ashvin Goel

Electrical and Computer Engineering
University of Toronto

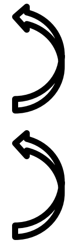
Distributed Systems
ECE419

Overview

- Introduction to byzantine fault tolerance
- Three Generals problem
- Primer on secure channels
- Practical byzantine fault tolerance

Review: distributed system models

- Network behavior
  - Reliable links: message received only if sent, may be reordered
  - Best-effort links: messages may be lost, duplicated, or reordered, with retries messages eventually gets through
  - Insecure links: adversary may eavesdrop, modify, drop messages



- Node behavior
  - Crash-stop failure: node crashes (e.g., power failure), stops forever
  - Crash-recovery failure: node crashes, resumes, disk data survives crash
  - Byzantine (fail-arbitrary) failure: node may execute incorrectly, including being malicious
- Timing behavior
  - Synchronous: message latency and node execution have bounds
  - Asynchronous: message latency and node execution have no bounds
  - Partially synchronous: system is most sync, occasionally async

Byzantine node behavior

- Until now, we have designed protocols assuming crash-stop or crash-recovery node behavior
  - E.g., for broadcast, state-machine replication
- Now we will look at how distributed systems can handle byzantine node behavior
  - Byzantine algorithms are very different because is not possible to convert one node behavior into another
- The term “Byzantine” is drawn from an allegory in the 1982 paper called “Byzantine Generals Problem”
  - It has no specific historical basis

Byzantine node failures

Byzantine node failures can occur due to

- Hardware faults
  - E.g., firmware bugs, bit flips in memory, corrupted n/w packets, etc.
- Software bugs
  - E.g., logic errors, memory corruption, concurrency bugs, etc.
- Malicious behavior
  - Run modified/arbitrary code on a node, don’t follow protocol
  - Send altered messages, insert messages, delay/drop messages
  - Send different messages to different nodes
  - Spoof messages (send using another identity)
  - Collude with other malicious nodes

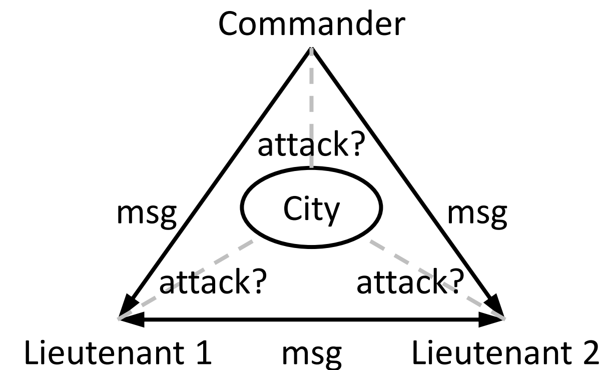
- In all these cases, we call node faulty, otherwise correct
  - Faulty nodes can produce arbitrary output (or no output)

Byzantine fault tolerance

- Byzantine fault tolerance is the ability of a system or service to tolerate (or survive under) byzantine faults
- Typically implemented using state machine replication, where some replicas may be faulty
  - Correct replicas agree to execute same operations, in same order
- Why not automatically detect, shutdown faulty replicas?
  - Faulty replicas may present different outputs to other replicas, e.g. correct output to some, wrong to others
  - Not always possible to know if replica is faulty from its output
- Let's start with a result that shows when byzantine fault tolerance is *not* possible

Three Generals Problem

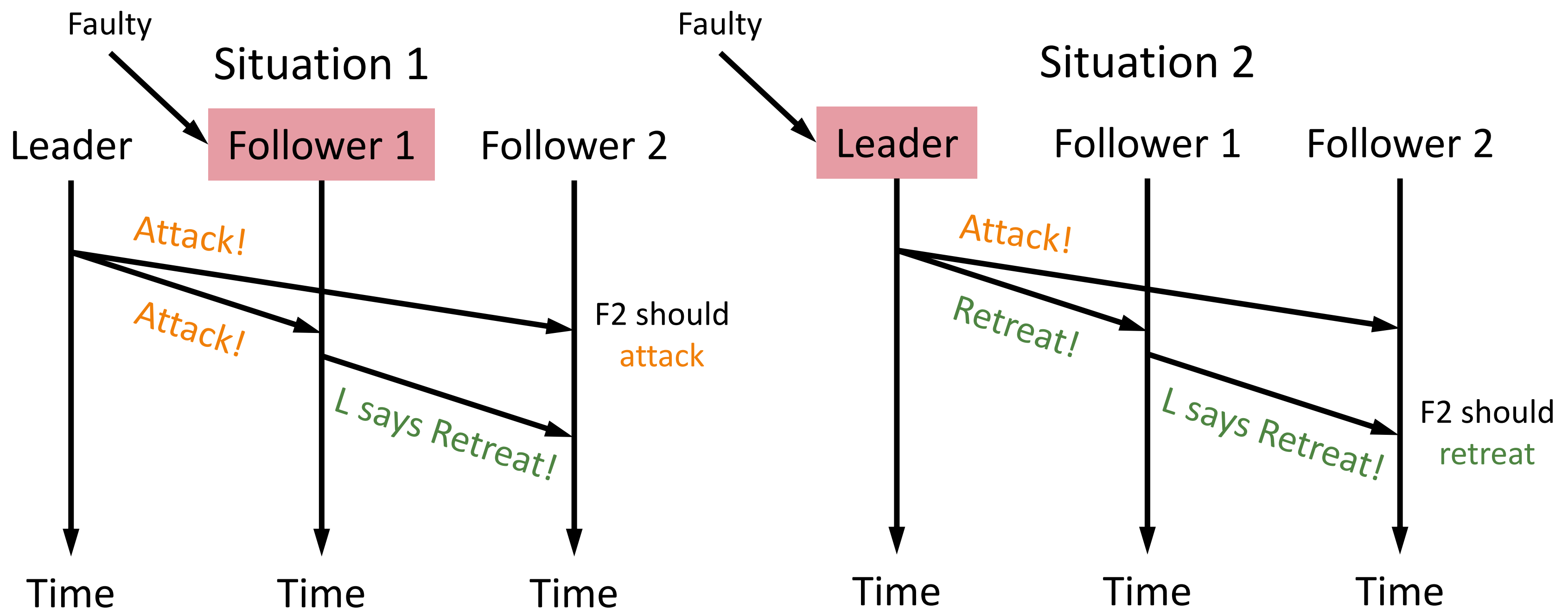
Three Generals problem



- A thought experiment that shows the challenge with coordinating actions, i.e., reaching agreement, when nodes are byzantine
  - Assume reliable link, synchronous timing
    - Message received unchanged, receiver know sender's identity
    - Dropped messages can be reliably detected
- Byzantine agreement problem
  - 1 leader, 2 followers, at most one of the three may be faulty
  - Leader sends *attack* or *retreat* order (message) to followers
  - Requirements
    - Reqr1: All correct followers must agree on same order
    - Reqr2: If leader is correct, correct followers must agree with leader

Three Generals dilemma

- For correct F2, Situations 1 and 2 are indistinguishable
  - Situation 1: Leader is correct, F2 should attack by Reqr2
  - Situation 2: Leader is faulty, F2 should retreat by Reqr1
- With 1 faulty node, 3 nodes cannot reach agreement!



Primer on secure channels

Why secure channels?

- Three Generals problem assumes a reliable link
- But followers can still lie about the leader's command
  - Makes it more complicated to solve byzantine failure problems
- This problem can be avoided if nodes can sign messages
- Signed messages allow followers to check
  1. Message originated at a leader, and
  2. Message has not been changed

Secure channels

- Sender encrypts message to ensure
  - Confidentiality: only intended receiver can decrypt message
- Sender signs message to ensure
  - Integrity: data is trustworthy, i.e., message hasn't been changed
  - Authentication: allows receiver to verify sender's identity
- We will discuss signing messages, needed for this lecture

Cryptographic hash

- A hash function H converts large input into small output
  - $h = H(m)$, m is message, h is called message digest (fixed size)
- We will use a cryptographic hash function with three properties to sign messages
  - Given h and H, it is hard to find m such that $h = H(m)$ } One way
  - Given m, it is hard to find m' such that $H(m) = H(m')$ } Second pre-image resistance
  - It is hard to find any m, m' such that $H(m) = H(m')$ } Collision resistance
- Crypto hashes, e.g., 128-bit MD5, 160-bit SHA-1, 256-bit SHA-2, are used for ensuring integrity, naming data, etc.

Message authentication codes

- A Message Authentication Code (MAC) use hashes for providing integrity and authentication
- Say Alice wants to send a signed message m to Bob
  - Assume k is secret key known only to Alice and Bob
- Alice constructs MAC as h=H(F(k, m)), F is function of H
- Alice sends [m, h] to Bob
- Bob verifies H(F(k, m)) is h
  - If so, Bob has high assurance:
    1. Whoever generated h must know key k, so m must have been generated by Alice (authenticates message sender)
    2. Message m has not been changed (message integrity)

MAC limitations

- MAC requires secret key to be known to Alice and Bob
  - But sharing a secret key securely is not simple
- An alternative is to use public-key cryptography

Public-key cryptography

- Every user owns a pair of keys, public and private key
  - User distributes the public key, often in a well-known location
  - User keeps the private key in a safe place
  - The private and public key reveal nothing about each other
- Message can be encrypted with either key,
  can only be decrypted with the other key
- Say Alice wants to send Bob message m securely
  - Alice encrypts m using Bob’s public key (pub(b)): c = E(pub(b), m)
  - Alice sends encrypted message c to Bob
  - Bob decrypt’s c using his private key (pri(b)): m = D(pri(b), c)

Digital signatures

- A digital signature is like a MAC but uses public-key cryptography for providing integrity and authentication
- Say Alice wants to send a signed message m to Bob
  - Assume Alice's public and private keys are: pub(a), pri(a)
- Alice constructs digital signature: sig(a) = E(pri(a), H(m))
- Alice sends [m, sig(a)] to Bob
- Bob verifies D(pub(a), sig(a)) is H(m)
  - If so, Bob has high assurance that:
    1. Whoever generated sig(a) must know key pri(a), so m must have been generated by Alice (authenticates message sender)
    2. Message m has not been changed (message integrity)

Practical Byzantine Fault Tolerance (PBFT)

Castro and Liskov, OSDI '99

Thanks to MIT 6.824 course notes

What is PBFT?

- Recall, Raft is a state machine replication protocol that provides fault tolerance under crash-recovery failures
  - Uses 2F+1 replicas, assumes F replicas may crash
  - Uses quorum of F+1 replicas for consensus
  - Handle crashed/delayed nodes, lost/delayed messages
- PBFT is a state machine replication protocol that provides fault tolerance under byzantine failures
  - Uses 3F+1 replicas, assumes F replicas may be faulty
  - Uses quorums of 2F+1 replicas for consensus
  - Must also handle malicious nodes, not so easy…
- Both Raft and PBFT ensure safety and liveness, linearizability under partially synchronous timing model

Attack model in PBFT

- An attacker can
  - Run arbitrary code on a faulty node
  - Control all F faulty nodes (and knows their crypto keys)
  - Temporarily delay a correct node
  - Can read any message
  - Temporarily delay any message, including between correct nodes
- An attacker cannot
  - Control more than F nodes
    - Requires node to have different implementations so they don’t have same bugs or vulnerabilities
  - Impersonate correct nodes, e.g., guess crypto keys of correct nodes, or break signed messages

Motivation for PBFT

- Consider two clients

```
Client 1:

put(config, “new config”)
put(config_done, TRUE)
```

```
Client 2:

while get(config_done) != TRUE:
    wait
get(config) // should be “new config”
```

- What could a faulty system do?
  - `get(config)` could return “old config” or totally random value
  - Ignore `put(config_done, TRUE)` or write `FALSE` value, Client 2 hangs
  - Perform `put(config_done, TRUE)` on some replicas, Client 2 hangs but Client 1 thinks `put` is done

PBFT setup

- Assume one or more clients, N servers (replicas)
  - F of N replicas can be faulty
  - All nodes have public-key pairs, know identities of other nodes
  - All nodes use digital signatures
    - Sender signs message, receiver authenticates message
- Basic protocol
  - Client sends a request to invoke an operation
  - Replicas execute operation, send reply to client with result
  - Client waits for result from replicas
- To understand PBFT, let's first try some simple designs

Try 1: Ask all

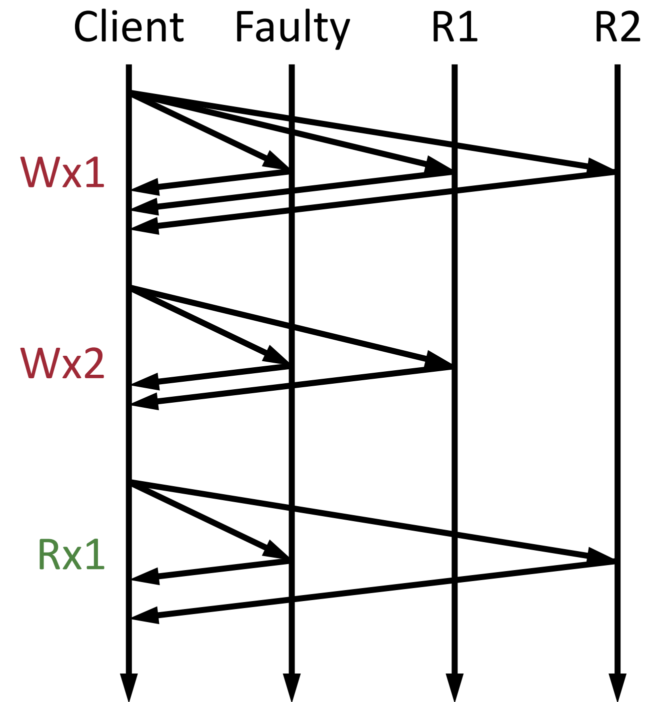
- Protocol
  - Client sends a request to invoke an operation to all replicas
    - Recall, requests must be deterministic for state machine replication
  - Replicas execute operation, send reply to client with result
  - Client waits for result from replicas
    - Request is successful if all N results match
- What could go wrong?
  - One replica is faulty, doesn't reply, or replies incorrectly
  - Stops progress

Try 2: Ask majority

- Liveness requirement
  - F replicas may be faulty => can only wait for at most N-F replies
- Assume N = 2F+1 replicas
- Protocol
  - Request is successful if N-F = F+1 results match
  - So, at least one result is from correct replica
- What could go wrong?
  - F+1 matching replies might be from F faulty replicas,
so maybe only one reply from correct replica
  - Next request also waits for F+1 replicas,
may not include the one correct replica of previous request,
so may not read latest data

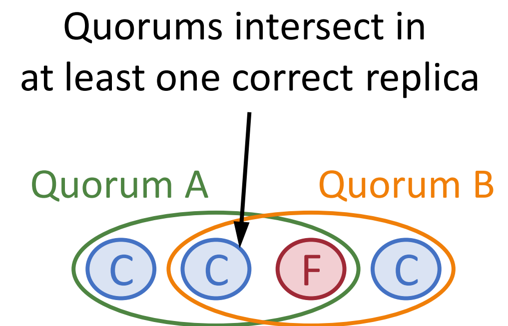
Try 2: Ask majority (2 out of 3)

- Client issues put(x, 1)
  - All replicas reply ok
- Client issues put(x, 2)
  - R2 misses request
  - 2 replicas reply ok
- Client issues get(x)
  - R1 misses request
  - 2 replicas reply 1
    - Faulty replica lies
  - Client reads stale data
    - Problem: put(x, 2) and get(x) have no common correct replica



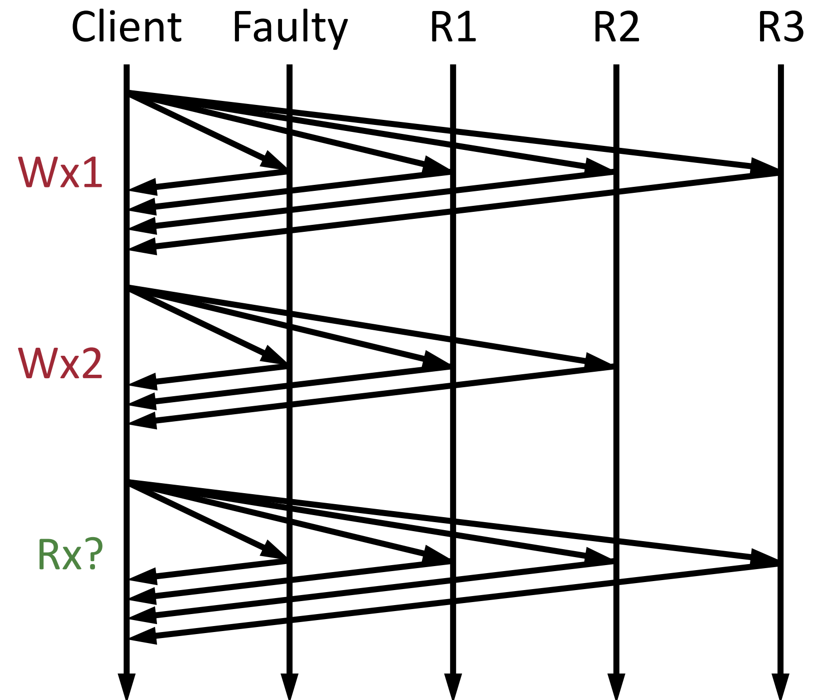
Try 3: Ask supermajority

- Liveness requirement
  - F replicas may be faulty => can only wait for at most N-F replies
- Quorum requirement
  - Of N-F replies, only N-2F replies may be from correct replicas
  - Need quorum intersection for any two requests to receive a reply from at least one common correct replica
    - N-2F correct replies must include a majority of correct replicas
    - N-2F > (N-F)/2 => N > 3F
- Assume N = 3F+1 replicas
- Protocol
  - Request is successful if N-F = 2F+1 results match
  - So, matching results from at least F+1 (majority of) correct replicas



Try 3: Ask supermajority (3 out of 4)

- Client issues put(x, 1)
  - All replicas reply ok
- Client issues put(x, 2)
  - R3 misses request
  - 3 replicas reply ok
- Client issues get(x)
  - R1 and R2 reply 2
  - Faulty and R3 reply 1
    - Faulty replica lies, R3 returns stale value
  - Client waits for 3 matching replies
    - Client can detect that there is a problem



With 3F+1 replicas,
2F+1 matching replies allows
handling F faulty replicas

Ordering requests

- Until now, we have assumed 1 client issues requests, but what about multiple clients issuing concurrent requests?
  - Correct replicas must process requests in same order
- Let's use a primary replica to pick an order
- But a primary replica can be faulty, so it can
  - Ignore a client request

    => Client may need to send requests to all replicas
  - Send requests to different replicas in different order

    => Replicas need to communicate with each other to ensure they received the same request in same order before executing request
  - Send incorrect result to client

    => Replicas need to directly send result to client

Try 4: Add a primary

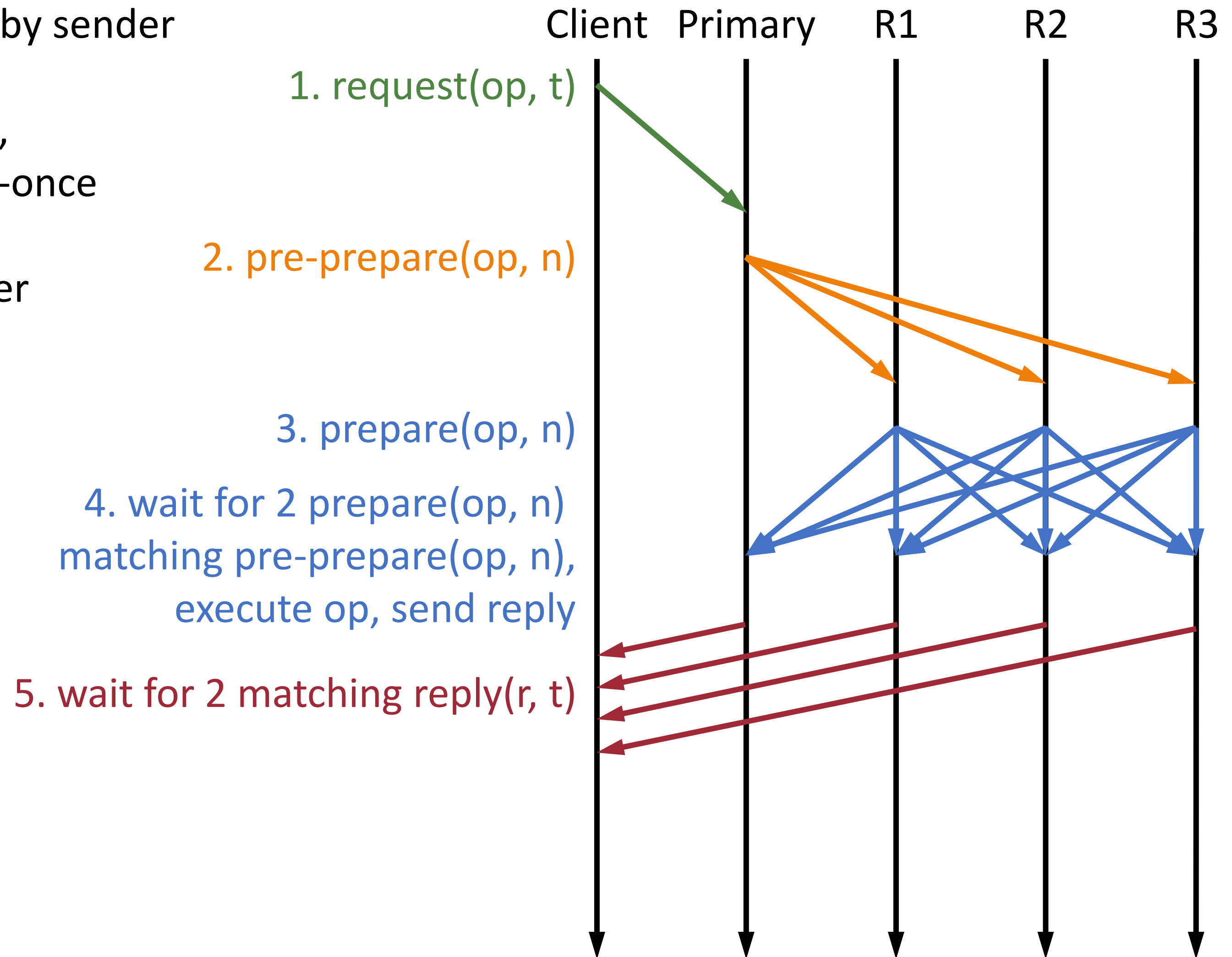
- Assume N = 3F+1 replicas, 1 is primary, others backup
- Protocol
  1. Clients send a request to invoke an operation op to primary
  2. Primary orders requests, assigns them sequence number n, sends PRE-PREPARE(op, n) message to all backups
  3. Each backup sends PREPARE(op, n) message to all replicas
  4. Each replica waits to receive matching PREPARE(op, n) from 2F+1 replicas (including self): ← Why 2F+1?
     - Replica executes operation (in sequence number order), sends reply to client with result
  5. Client waits for result from replicas
     - Request is successful if F+1 results match ← Why F+1?

Try 4: Add a primary (F = 1, N = 4)

All messages signed by sender

t is client timestamp,
helps ensure exactly-once

n is sequence number



What about correctness, progress?

- Can replicas modify/forge client's request?
  - Client signs request, so attack is detectable
- What if faulty backups drop or delay their messages?
  - If primary is correct, it sends same request to all replicas, protocol progresses since replicas wait for 2F+1 matching prepares
- What if primary drops or delays requests?
  - If a client does not receive a reply for a request in time, it resends its request to all replicas
    - Backups relay the request to the primary
  - If a backup receives this client request and times out waiting to execute requested operation, it suspects primary
    - When enough backups suspect primary, they choose another primary

What about correctness, progress?

- How else can replicas misbehave?
- What if primary sends requests in different order?
  - If F+1 or more correct replicas get 2F+1 matching prepares:
    - These replicas receive and execute the same requested operation, client gets enough matching replies, protocol makes progress
    - Rest of the correct replicas wait, but will not get 2F+1 matching prepares for some other request, may ask to change primary
  - Otherwise:
    - Client waits, protocol make no progress
    - F or less correct replicas may execute the requested operation, but operation may never be successful (execute at F+1 correct replicas)
      - We will fix that soon
    - Client, backups take same action as when primary drops or delays requests (previous slide)

Choosing new primary

- As we have seen, a faulty primary can stop progress
- How to choose a new primary?
  - Need to ensure faulty replicas don't always become primary!
    - Elections can be subverted by faulty replicas colluding, denying service
- Use a round-robin protocol for choosing a primary
  - Let's divide the protocol into a sequence of views
    - Views are numbered sequentially, i.e., v = 0, 1, 2, 3, …
    - Replicas are numbered sequentially, i.e., r = 0, …, N-1
  - Each view has one primary replica, rest are backup
    - Primary in View v is Replica r, where r = v mod n
  - No more than F faulty replicas in a row, ensures progress

View change

- Backups ask to change primary (view change) when they timeout waiting to execute an operation
- Protocol
  - Backups send VIEW-CHANGE message to new primary
  - New primary waits for enough VIEW-CHANGE messages
    - We will discuss how many soon
  - New primary sends NEW-VIEW message to all replicas with
    - All VIEW-CHANGE messages it received to prove that enough replicas asked for a view change
  - New primary numbers requests after last operation it executed

View change problem

- Need to ensure that all correct replicas agree about request numbers across view change
- Problem
  - Correct replica saw 2F+1 PREPAREs for request n, executed it
  - New primary executed operation n-1, hasn't even seen request n
  - New primary starts numbering at n, two different requests at n
- Can new primary ask all correct replicas for operations they have executed?
  - No, new primary can only wait for 2F+1 matching replies, not all 2F+1 correct replicas!
  - How can new primary learn about already executed requests?

View change solution

- Idea: a replica should let enough replicas know that it is prepared to execute an operation so new primary can learn about this operation
- Basic solution
  - When a replica receives 2F+1 PREPARE for a request, we will say replica/request is prepared
  - A replica should execute an operation only after it knows that a majority of correct replicas are prepared
  - Requires a third COMMIT phase in the protocol
  - As we will see, new primary can then learn about any prepared request at any replica by asking a majority of correct replicas
- We are finally ready to see the PBFT protocol!

PBFT protocol

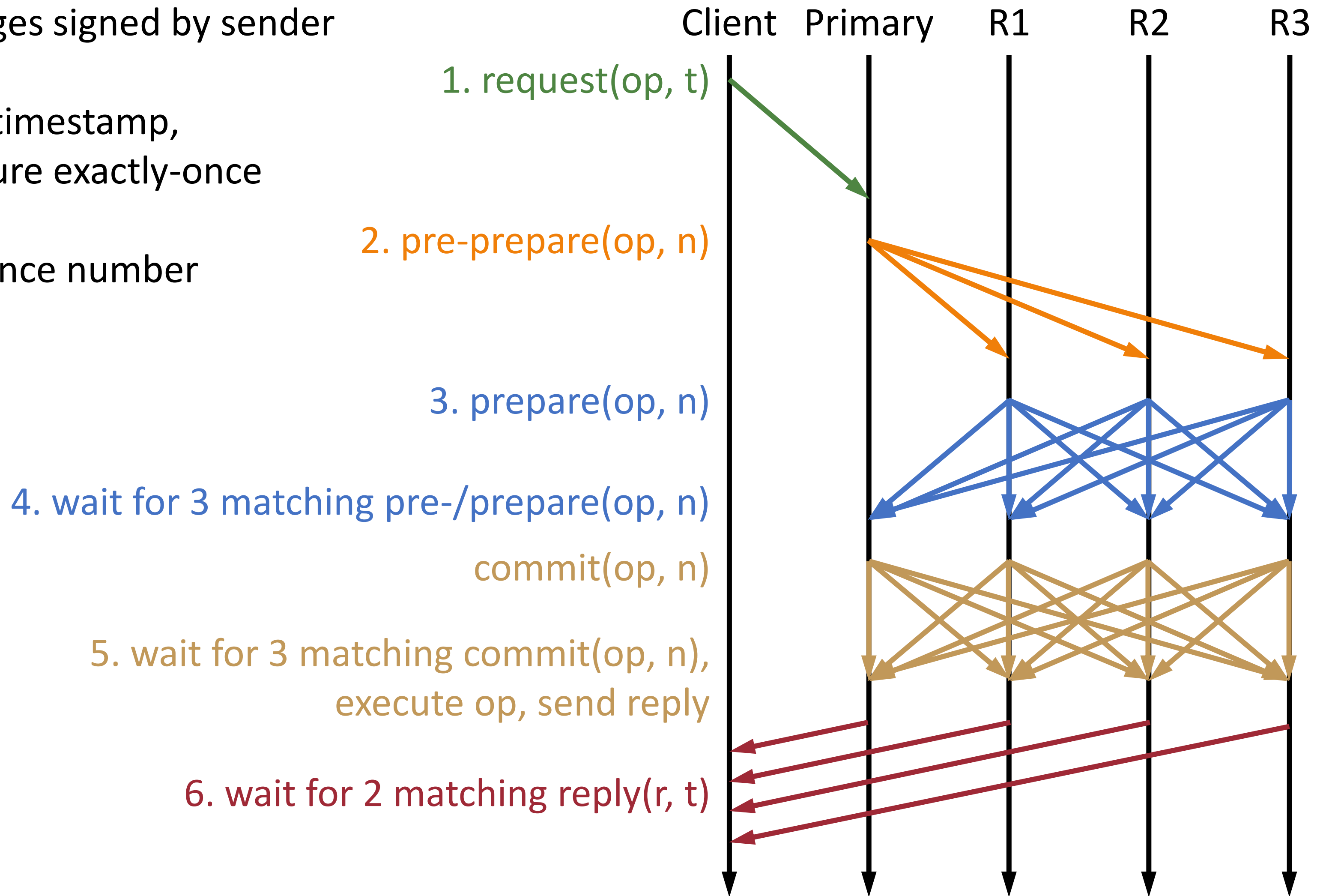
1. Clients send a request to invoke an operation op to primary
2. Primary orders requests, assigns them sequence number n, sends PRE-PREPARE(op, n) message to all backups
3. Each backup sends PREPARE(op, n) message to all replicas
4. Each replica waits to receive matching PREPARE(op, n) from 2F+1 replicas (including self):
   - Replica sends COMMIT(op, n) to all replicas
5. Each replica waits to receive matching COMMIT(op, n) from 2F+1 replicas (including self):
   - At least F+1 correct replicas are prepared to execute op (committed)
   - Replica executes operation (in sequence number order), sends reply to client with result
6. Client waits for result from replicas
   - Request is successful if F+1 results match

PBFT (F = 1, N = 4)

All messages signed by sender

t is client timestamp,
helps ensure exactly-once

n is sequence number



PBFT view change protocol

- Backups ask to change primary (view change) when they timeout waiting to execute an operation
- Protocol
  - Backups send VIEW-CHANGE message to new primary
    with recent prepared requests, each with 2F+1 PREPARE messages
  - New primary waits for 2F+1 VIEW-CHANGE messages
  - New primary sends NEW-VIEW message to all replicas with
    - Complete set of VIEW-CHANGE messages to prove that
      a majority of correct replicas asked for a view change
    - List of all prepared requests received in any VIEW-CHANGE,
      so that replicas can execute, if needed, all these requested operations

Correctness of PBFT view change

- Say a replica executes operation in request R,
  will the new primary know about it?
- Informal proof
  - Replica executes operation in prepared request R after it receives COMMIT from F+1 correct replicas, i.e., replica knows that majority of correct replicas have prepared request R
  - Primary waits for view-change from majority of correct replicas
  - At least one correct replica must have the prepared request R and will tell primary about this request
- Can the new primary ignore request R?
  - No, the 2F+1 VIEW-CHANGE messages are signed,
    replicas validate them when they receive them in NEW-VIEW,
    will receive, commit and execute request R

Summary of PBFT protocol

- Normal operation, after primary receives request:
  - PRE-PREPARE: primary initiates consensus by sending message to backups
  - PREPARE: backups send messages to all, replicas agree on order of request (within a view)
  - COMMIT: replicas send messages to all, replicas agree to commit request (across views)
- View change, after backups timeout:
  - VIEW-CHANGE: backups initiate consensus by sending message to new primary
  - NEW-VIEW: replicas agree on new primary and starting request number in new view

More details in PBFT paper

- Logging of messages so correct replicas can recover
- Checkpoints to garbage collect logs
- Cryptographic optimizations
- Communication optimizations to reduce size and latency of messages in common case
- Fast, one round-trip, read-only operations

Performance

- Request latency until commit is two round trips
- Number of messages is $O(N^2)$, where N is # of replicas
- Why is it called practical?
  - Ensures correctness
  - Ensures liveness under partially synchronous setting
  - Optimizations enable good performance

Applications of BFT

- BFT is not widely-used today
  - People rely on prevention, detection of compromised nodes
- BFT is seeing a revival in Blockchain systems
  - IBM's Hyperledger is a permissioned blockchain that uses PBFT
  - Stellar generalizes PBFT for federated deployments

Conclusions

- With byzantine failure, node may execute arbitrary code
- PBFT implements byzantine fault tolerance,
  i.e., state machine replication under byzantine failures
  - Requires 3F+1 nodes to handle F faulty nodes, optimal
  - Uses quorums of 2F+1 nodes for consensus,
    i.e.., to ensure a total order of requests within and across views
- Limitations
  - Requires independent node implementations
  - Identity and number of replicas must be known to all,
    typically assigned by a central authority
- Next, let’s look at systems that avoid these limitations