

Byzantine Fault Tolerance



جامعة الملك عبد الله
للعلوم والتقنية
King Abdullah University of
Science and Technology

CS 240: Computing Systems and Concurrency
Lecture 14

Marco Canini

So far: Fail-stop failures

- Traditional state machine replication tolerates **fail-stop failures**:
 - Node crashes
 - Network breaks or partitions
- State machine replication with $N = 2f+1$ replicas can tolerate **f simultaneous fail-stop failures**
 - Two algorithms: Paxos, RAFT

Byzantine faults

- **Byzantine fault:** Node/component **fails arbitrarily**
 - Might perform **incorrect computation**
 - Might give **conflicting information** to different parts of the system
 - Might **collude** with other failed nodes
- Why might nodes or components fail arbitrarily?
 - **Software bug** present in code
 - **Hardware failure** occurs
 - **Hack** attack on system

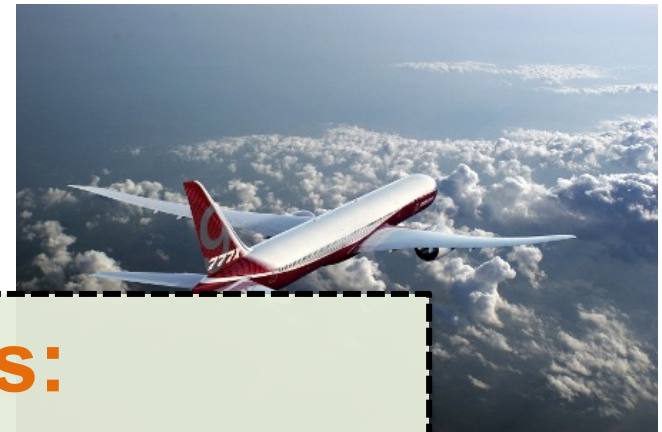
Today: Byzantine fault tolerance

- Can we provide state machine replication for a service **in the presence of Byzantine faults?**
- Such a service is called a **Byzantine Fault Tolerant (BFT)** service
- *Why might we care about this level of reliability?*

Mini-case-study: Boeing 777 fly-by-wire primary flight control system

- **Triple-redundant, dissimilar** processor hardware:

1. Intel 80486
2. Motorola
3. AMD

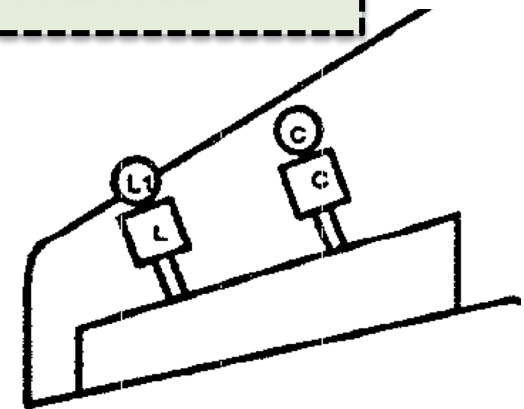


Key techniques:

- Each processor has **Hardware and software diversity** from the others
- **Voting** between components

Simplified design:

- Pilot inputs → three processors
- Processors **vote** → control surface



LEFT ELEVATOR

Today

1. **Traditional state-machine replication for BFT?**
2. Practical BFT replication algorithm
3. Performance and Discussion

Review: Tolerating one fail-stop failure

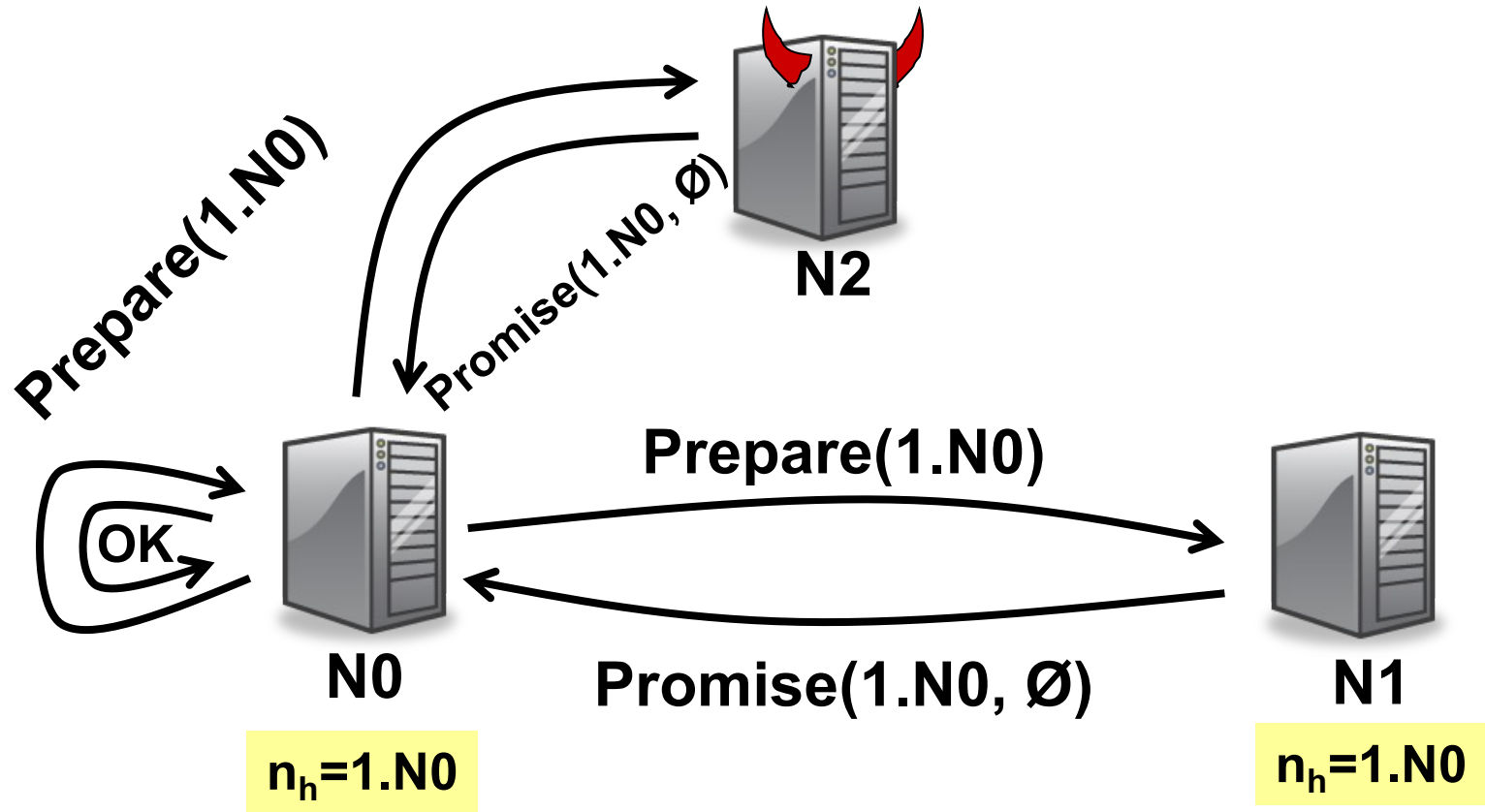
- **Traditional state machine replication (Paxos)** requires, e.g., $2f + 1 = \text{three}$ replicas, if $f = 1$
- Operations are totally ordered \rightarrow correctness
 - A two-phase protocol
- Each operation uses $\geq f + 1 = 2$ of them
 - **Overlapping** quorums
 - So at **least one replica** “remembers”

Use Paxos for BFT?

1. **Can't rely on the primary** to assign proposal #
 - Could assign same proposal # to different requests
2. **Can't use Paxos** for view change
 - Under Byzantine faults, the intersection of two majority ($f + 1$ node) quorums **may be bad node**
 - Bad node tells **different** quorums **different things!**
 - e.g. tells N0 accept **val1**, but N1 accept **val2**

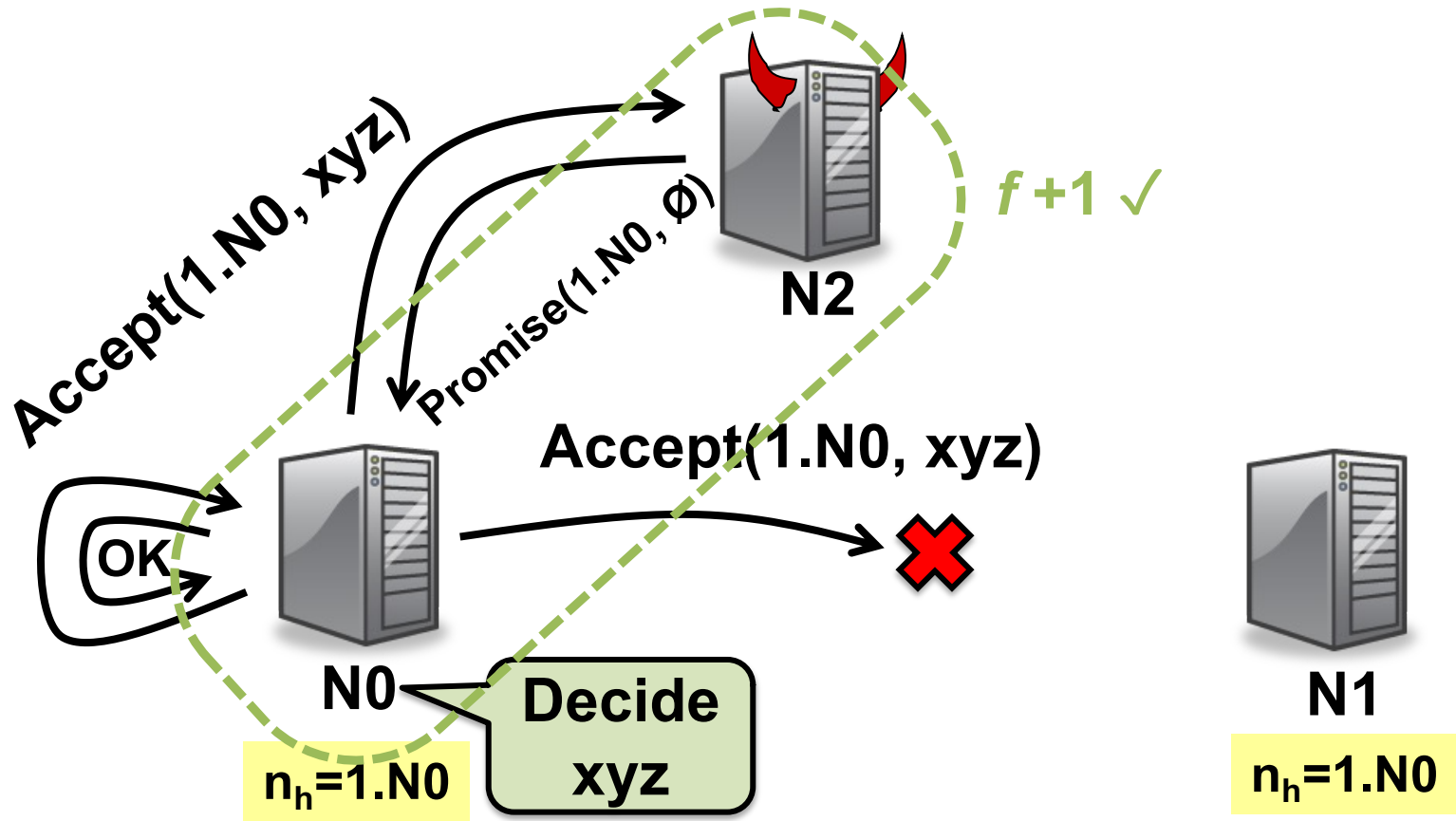
Paxos under Byzantine faults

($f=1$)



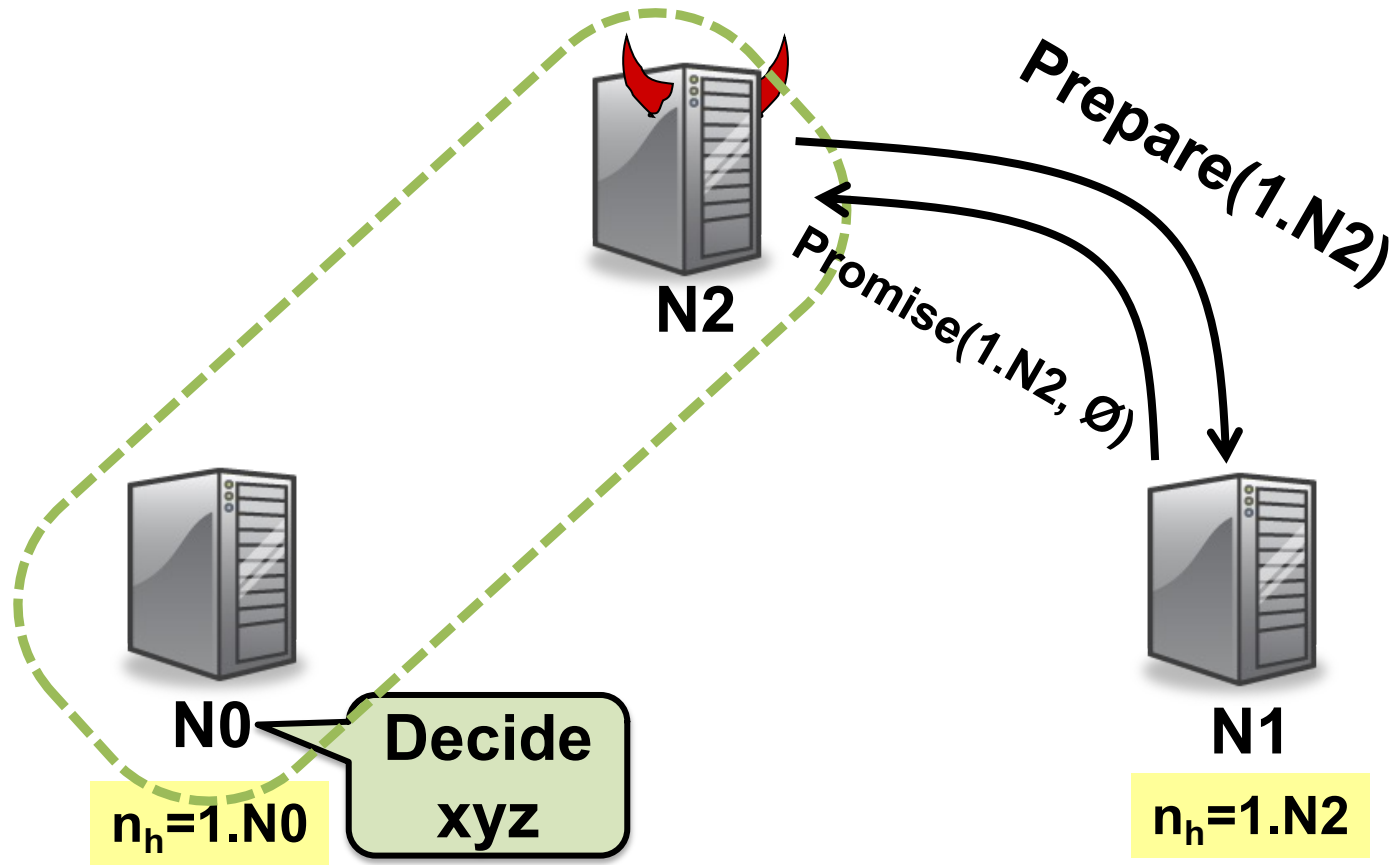
Paxos under Byzantine faults

($f = 1$)



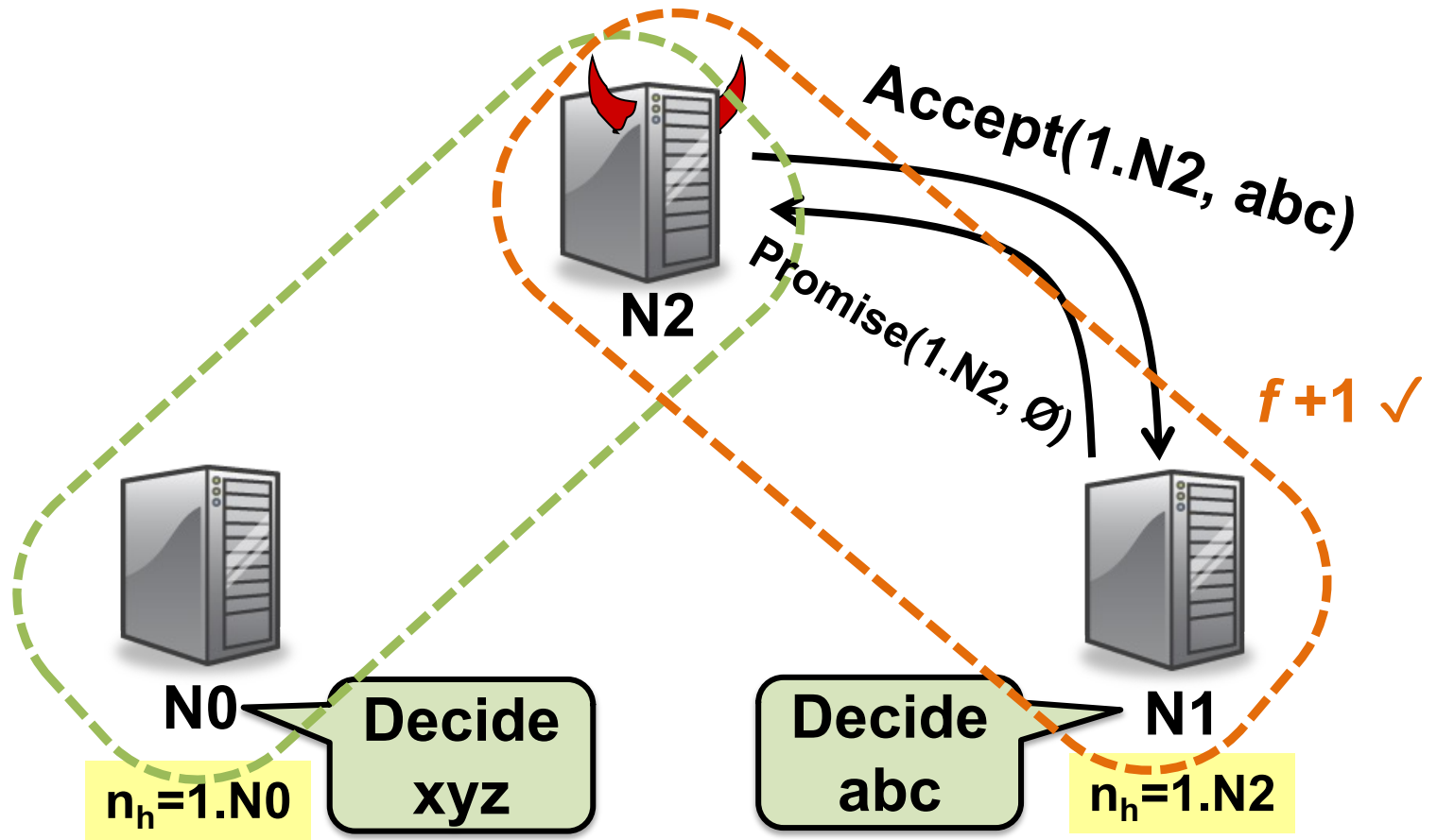
Paxos under Byzantine faults

($f=1$)



Paxos under Byzantine faults

($f = 1$)



Conflicting decisions!

Theoretical fundamentals: Byzantine Generals



General #1



General #2



Unreliable
messenger



General #3

Result: Using messengers, problem solvable iff $> \frac{2}{3}$ of the generals are loyal

Put burden on client instead?

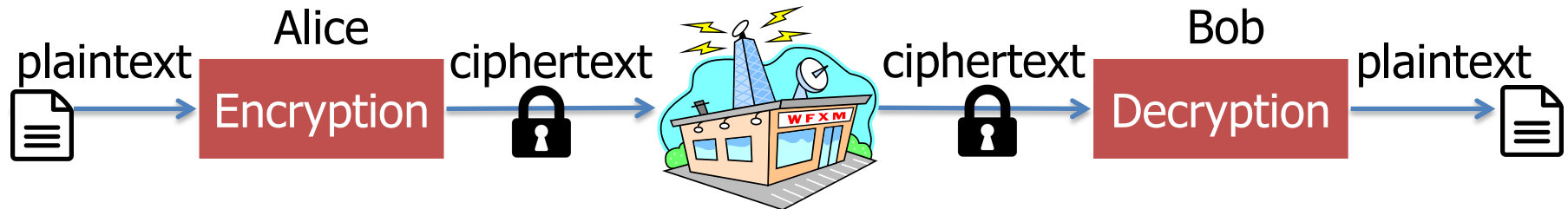
- Clients **sign** input data before storing it, then **verify** signatures on data retrieved from service
- **Example:** Store signed file $f_1 = \text{“aaa”}$ with server
 - Verify that returned f_1 is correctly signed

<cryptography in 6 slides>

κρυπτό + γραφή (Cryptography)

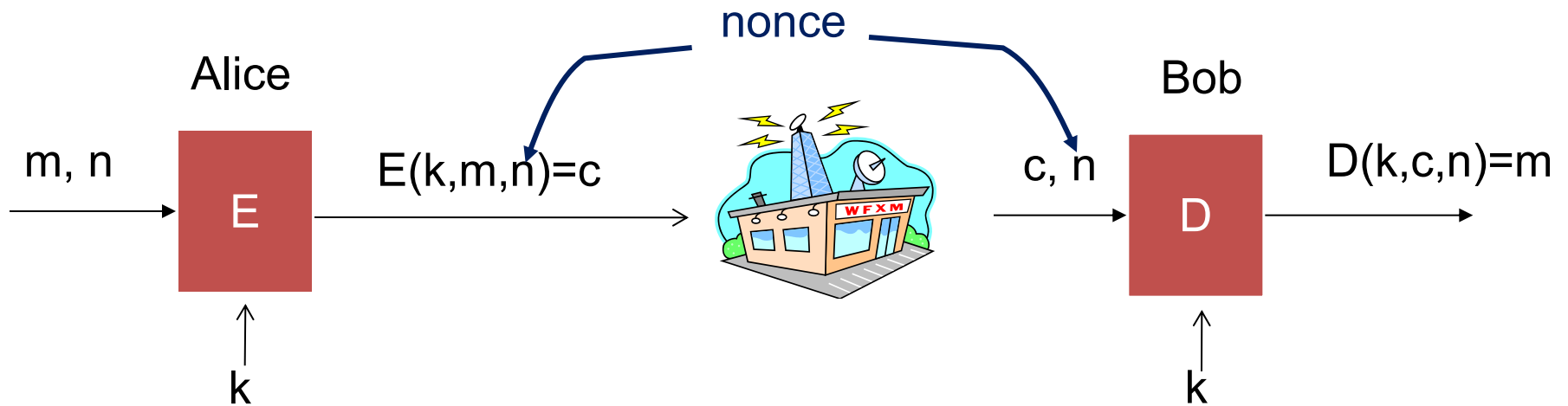
- Greek for “secret writing”
- **Confidentiality**
 - Obscure a message from eavesdroppers
- **Integrity**
 - Assure recipient that the message was not altered
- **Authentication**
 - Verify the identity of the source of a message
- **Non-repudiation**
 - Convince a 3rd party that what was said is accurate

Terminology



- Encryption algorithm
 - Transforms a **plaintext** into a **ciphertext** that is **unintelligible for non-authorized parties**
 - Usually parametrized with a cryptographic key
- **Asymmetric (Public) key cryptography**
 - Crypto system: encryption + decryption algorithms + key generation
- **Symmetric (Shared) key cryptography**
 - Cipher/decipher: symmetric-key encryption/decryption algorithms

Symmetric key encryption



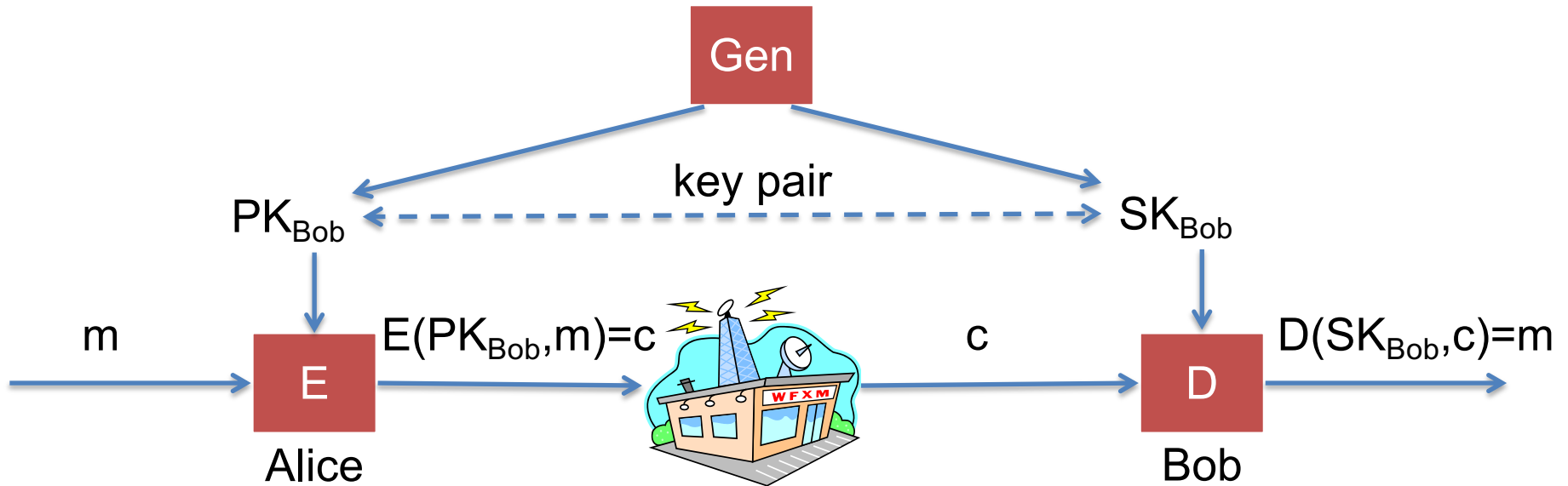
E, D: cipher k: secret key (e.g. 128 bits)

m, c: plaintext, ciphertext n: nonce (aka IV)

Encryption algorithm is **publicly known**

- Never use a proprietary cipher

Public key encryption



PK: public key , SK: secret key (e.g., 1024 bits)

Example: Bob generates (PK_{Bob}, SK_{Bob}) and gives PK_{Bob} to Alice

Applications

- Public-key encryption
 - Alice public key for encryption
 - Anyone can send encrypted message
 - Only Alice can decrypt messages (with secret key)

- Digital signature scheme
 - Alice public key for verifying signatures
 - Anyone can check a message signed by Alice
 - Only Alice can sign messages (with secret key)

Establishing a shared secret

Alice

$(pk, sk) \leftarrow G()$

“Alice”, pk

Bob

choose random
 x

“Bob”, $c \leftarrow E(pk, x)$

$D(sk, c) \rightarrow x$

x shared secret

</cryptography in 6 slides>

Put burden on client instead?

- Clients **sign** input data before storing it, then **verify** signatures on data retrieved from service
- **Example:** Store signed file $f_1 = \text{“aaa”}$ with server
 - Verify that returned f_1 is correctly signed

But a Byzantine node can **replay stale**, signed **data** in its response

Inefficient: Clients have to perform computations and sign data

Today

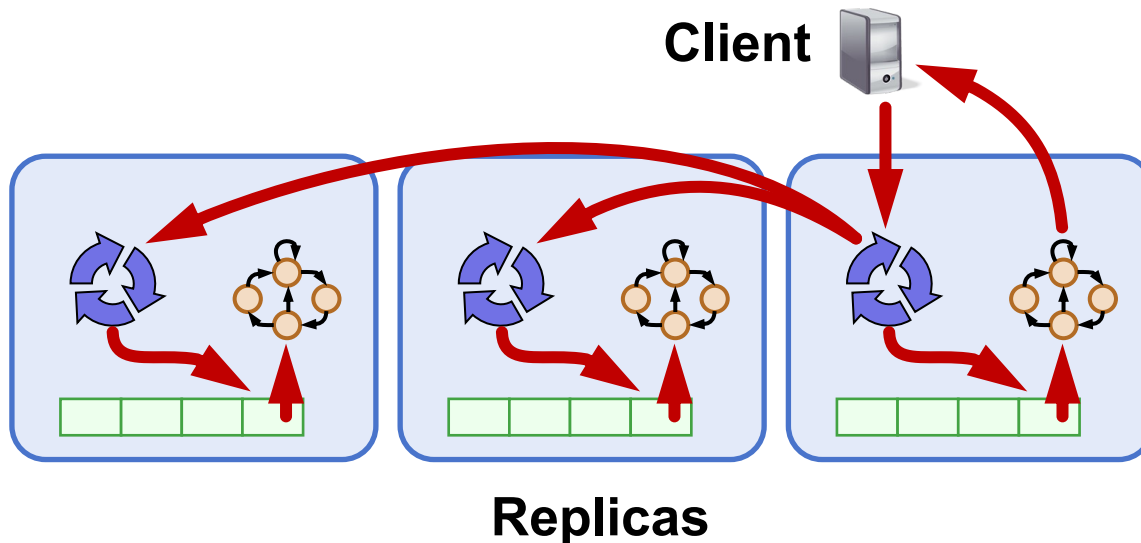
1. Traditional state-machine replication for BFT?
- 2. Practical BFT replication algorithm**
[Liskov & Castro, 2001]
3. Performance and Discussion

Practical BFT: Overview

- Uses $3f+1$ **replicas** to survive f **failures**
 - Shown to be **minimal** (Lamport)
- Requires **three phases** (not two)
- Provides **state machine replication**
 - Arbitrary service accessed by operations
 - E.g., file system ops read and write files and directories
 - **Tolerates** Byzantine-faulty clients

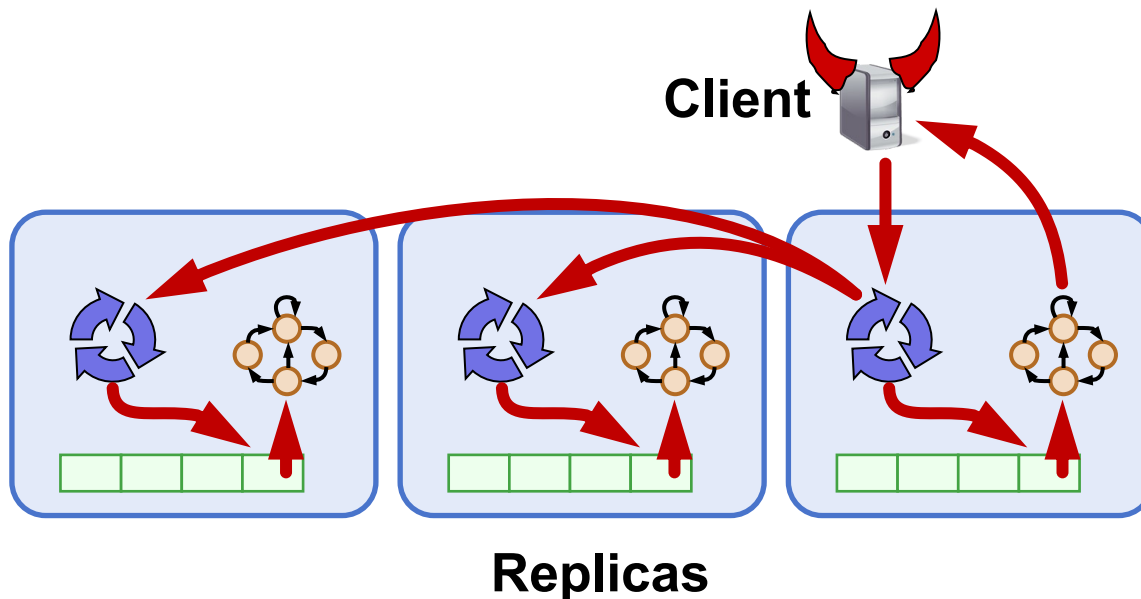
Correctness argument

- Assume operations are **deterministic**
- Assume replicas **start in same state**
- If replicas execute **same requests in same order**:
 - Correct replicas will produce **identical results**



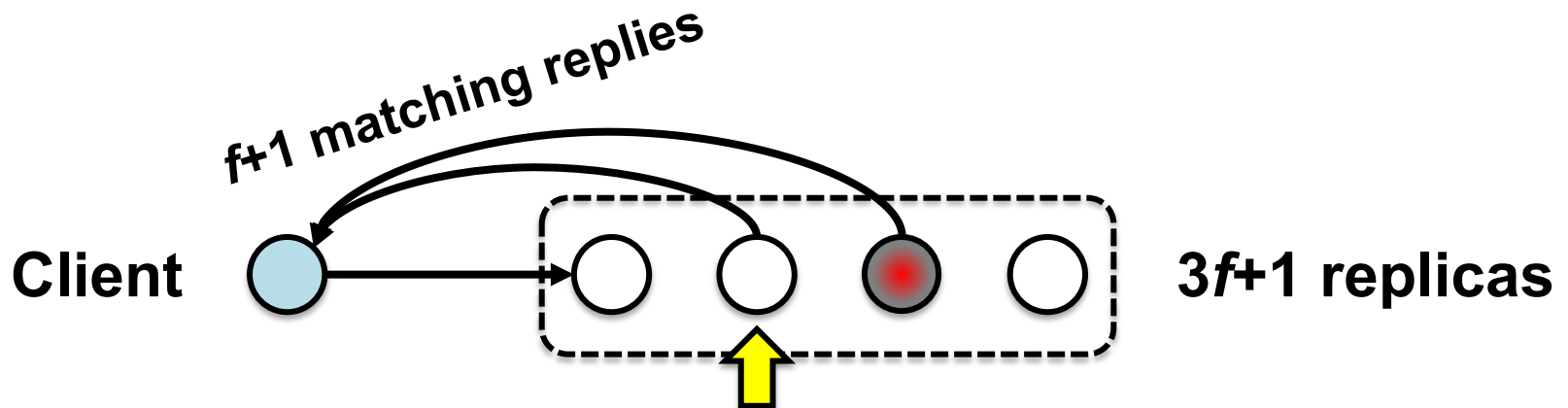
Non-problem: Client failures

- Clients **can't** cause replica inconsistencies
- Clients **can** write bogus data to the system
 - **Sol'n:** Authenticate clients and separate their data
 - This is a **separate problem**



What clients do

1. Send requests to the primary replica
 2. Wait for $f+1$ **identical** replies
 - **Note:** The replies may be deceptive
 - *i.e.* replica returns “correct” answer, but locally does otherwise!
- But \geq **one** reply is **actually** from a **non-faulty replica**

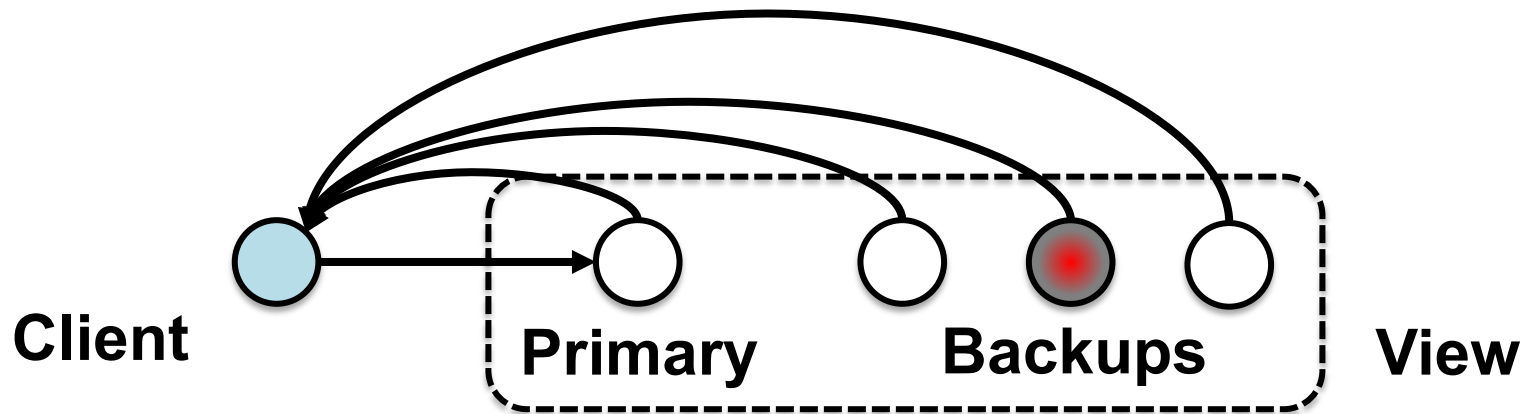


What replicas do

- Carry out a protocol that ensures that
 - Replies from honest replicas are correct
 - Enough replicas process each request to ensure that
 - The **non-faulty** replicas process the **same requests**
 - In the **same order**
- Non-faulty replicas obey the protocol

Primary-Backup protocol

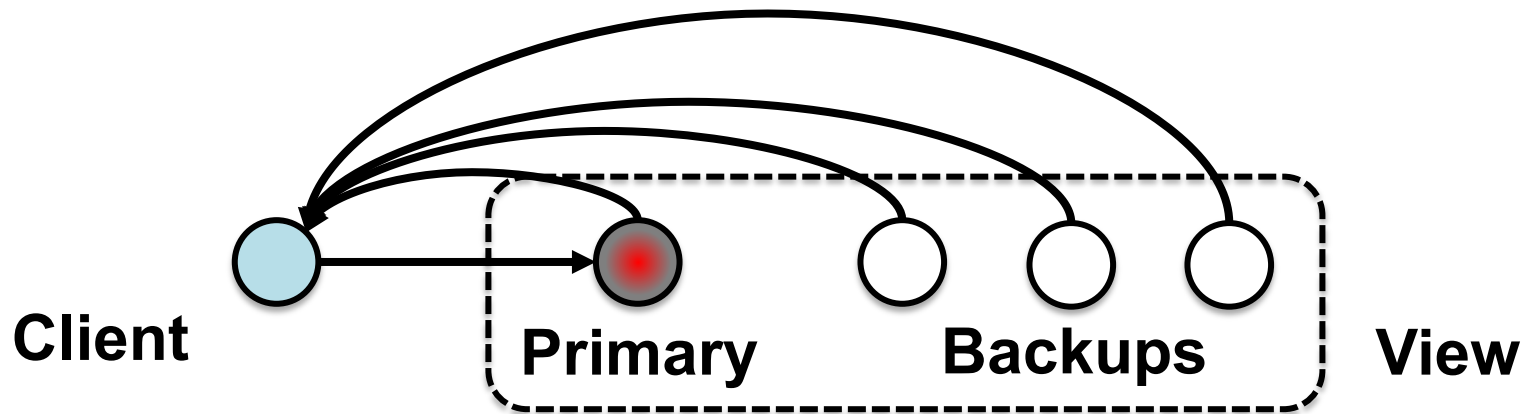
- Primary-Backup protocol: Group runs in a **view**
 - View **number** designates the **primary** replica



- Primary is the node whose **id (modulo view #) = 1**

Ordering requests

- Primary picks the ordering of requests
 - But the **primary might be a liar!**

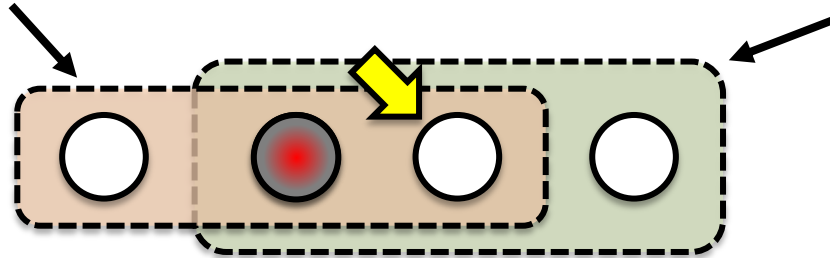


- Backups ensure primary behaves correctly
 - Check and certify correct ordering
 - Trigger **view changes** to replace faulty primary

Byzantine quorums

($f = 1$)

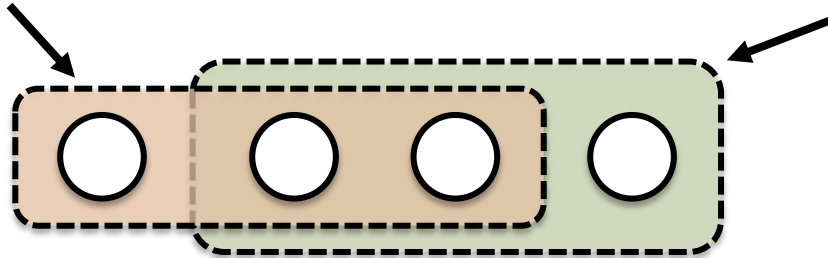
A *Byzantine quorum* contains $\geq 2f+1$ replicas



- One op's quorum **overlaps** with next op's quorum
 - There are $3f+1$ replicas, in total
 - **So overlap is $\geq f+1$ replicas**
- $f+1$ replicas must contain ≥ 1 **non-faulty replica**

Quorum certificates

A *Byzantine quorum* contains $\geq 2f+1$ replicas



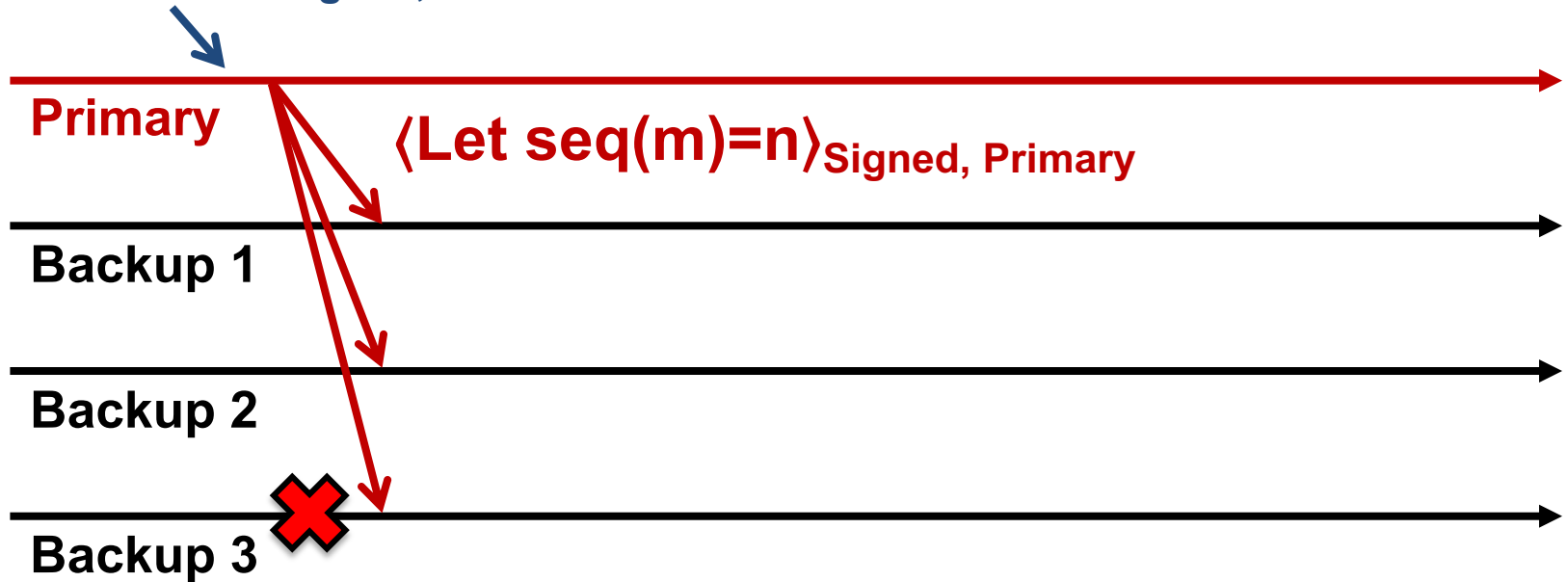
- ***Quorum certificate***: a collection of $2f + 1$ signed, **identical** messages from a Byzantine quorum
 - All messages agree on the **same statement**

Keys

- Each client and replica has a **private-public keypair**
- **Secret keys:** symmetric cryptography
 - Key is known only to the two communicating parties
 - Bootstrapped using the public keys
- **Each client, replica** has the following secret keys:
 - One key per node for sending messages
 - One key per node for receiving messages

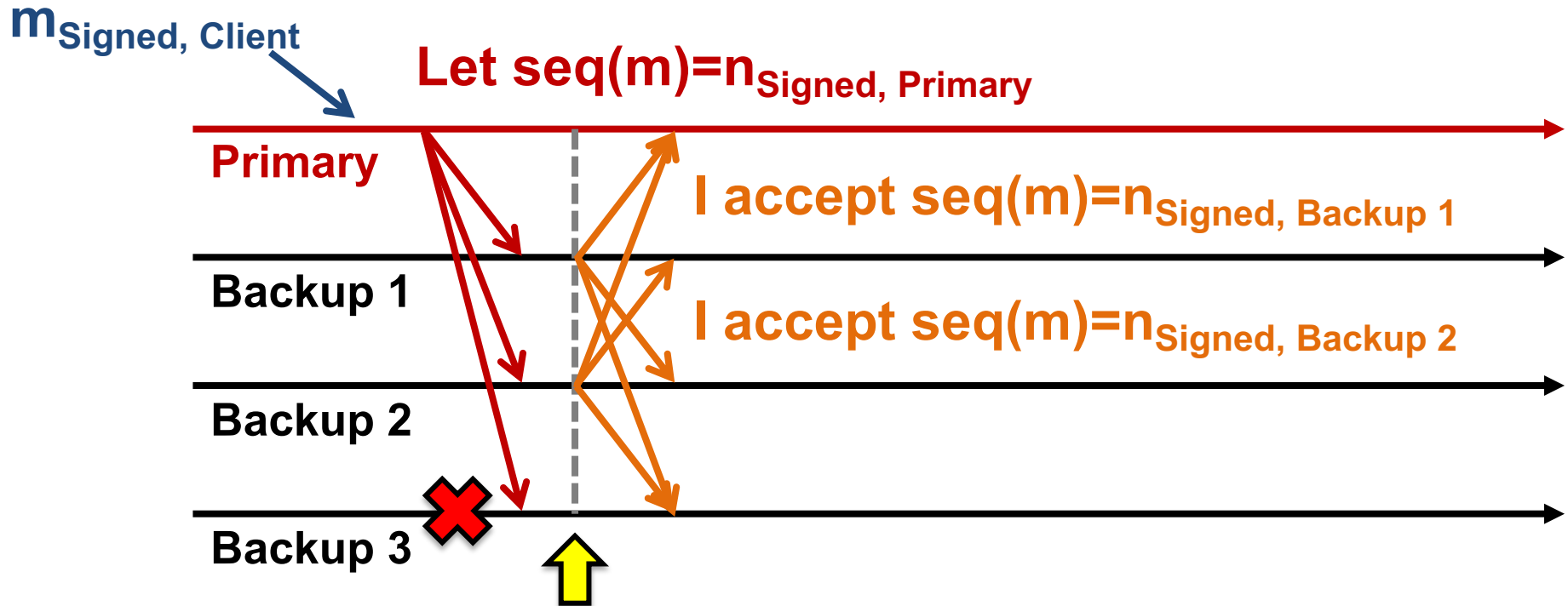
Ordering requests

$m = \langle \text{request}, \text{op}, t \rangle_{\text{Signed, Client}}$



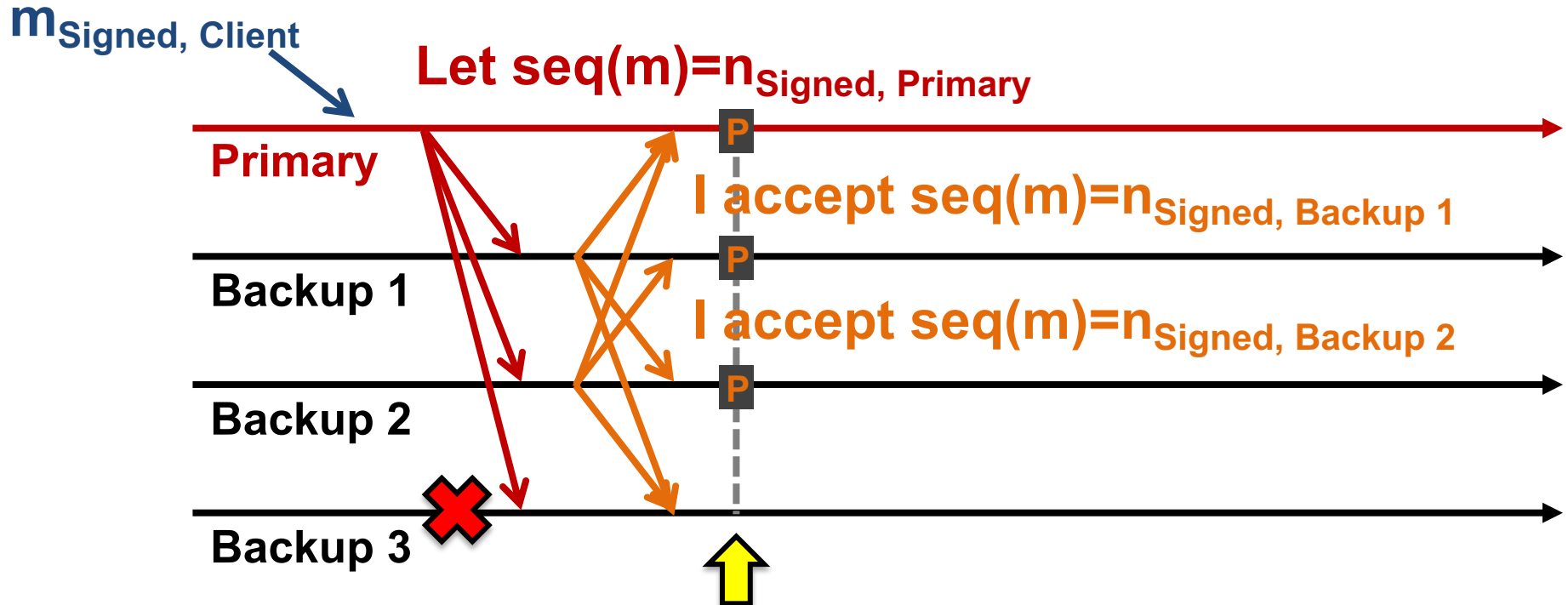
- Client requests operation **op** with *timestamp* t
- Primary chooses the request's *sequence number* (n)
 - Sequence number determines order of execution

Checking the primary's message



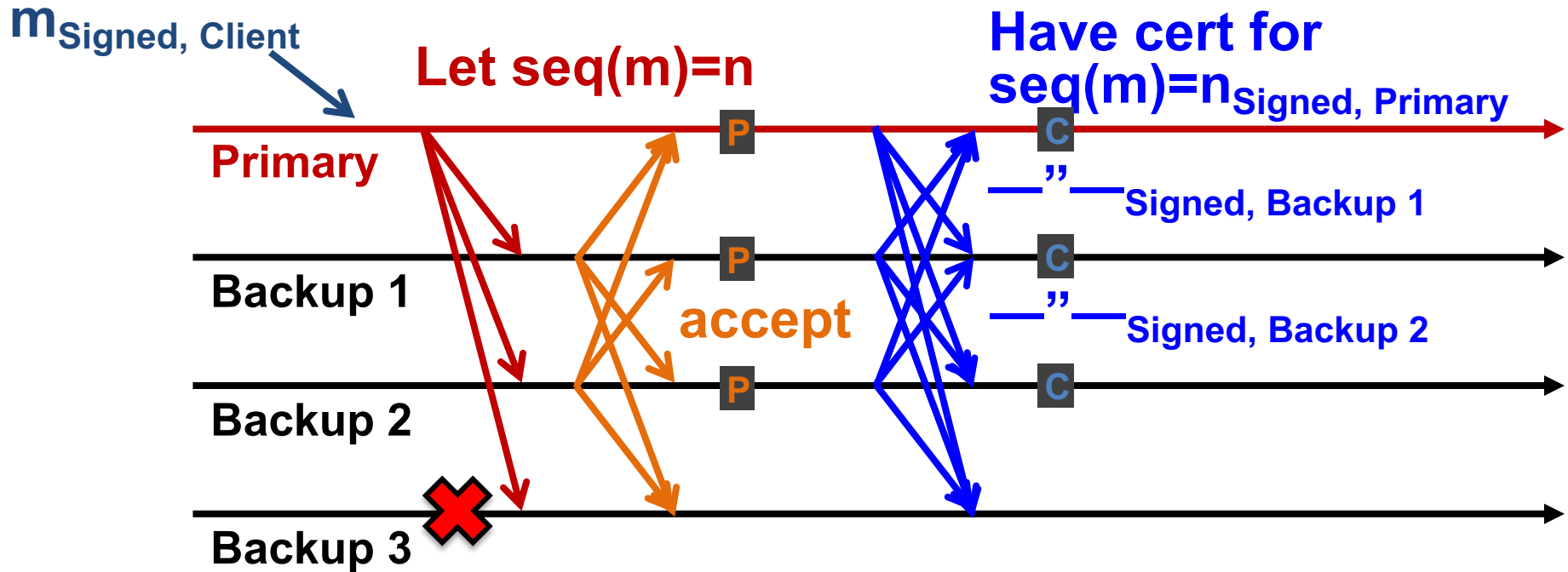
- Backups **locally** verify they've seen \leq **one** client request for sequence number n
 - If local check passes, replica broadcasts **accept** message
 - Each replica makes this decision **independently**

Collecting a *prepared certificate* ($f=1$)



Each **correct** node has a prepared certificate **locally**, but does not **know** whether the **other correct nodes** do too! So, we **can't commit** yet!

Collecting a *committed* certificate ($f = 1$)



Once the request is **committed**, replicas **execute** the operation and send a reply directly back to the client.

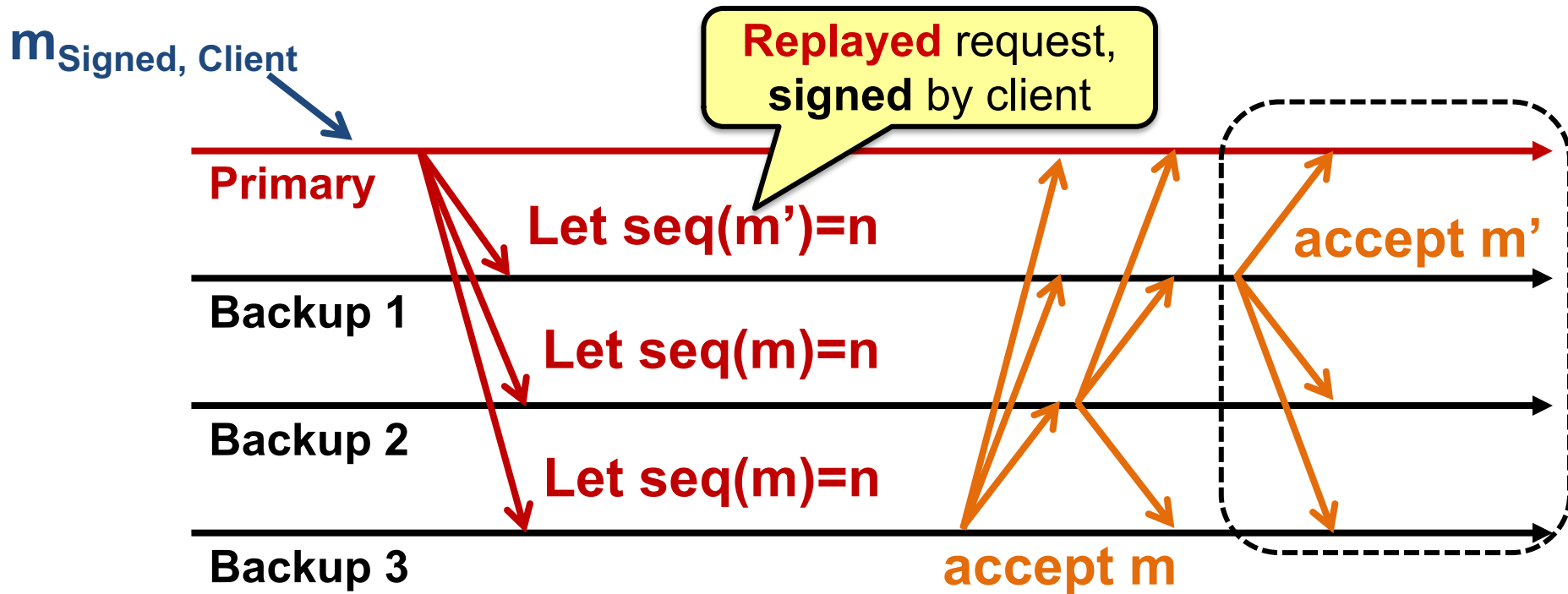
Byzantine primary: replaying old requests

- The client assigns each request a unique, monotonically increasing *timestamp* t
- Servers track greatest t executed for each client c , $T(c)$, and their corresponding reply
 - On receiving request to execute with timestamp t :
 - If $t < T(c)$, skip the request execution
 - If $t = T(c)$, resend the reply but skip execution
 - If $t > T(c)$, execute request, set $T(c) \leftarrow t$, remember reply

Malicious primary can invoke $t = T(c)$ case but **cannot compromise safety**

Byzantine primary: Splitting replicas

($f = 1$)

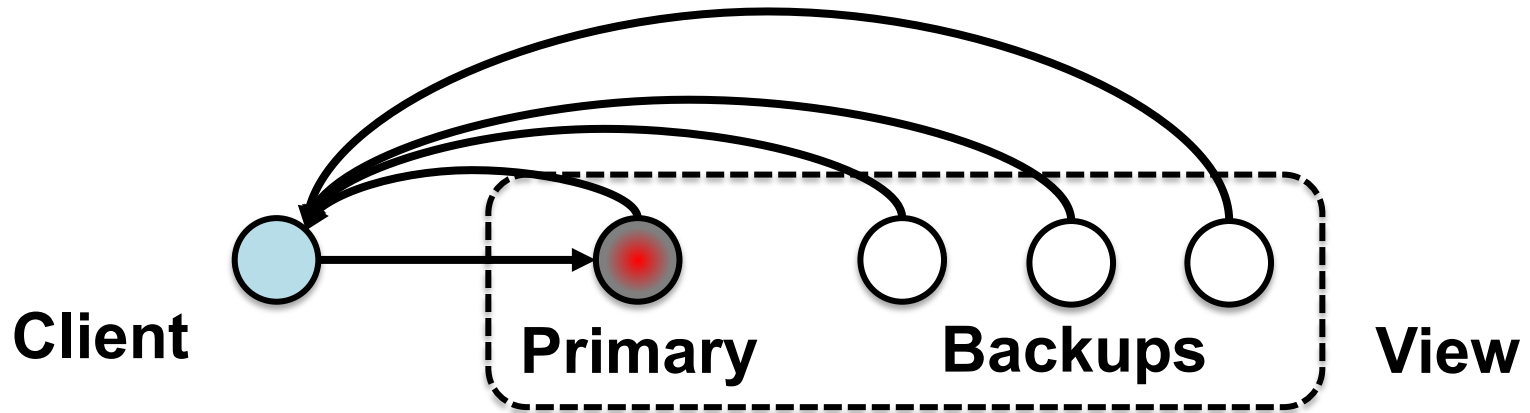


- **Recall:** To **prepare**, need primary message and $2f$ accepts
 - Backup 1: **Won't prepare m'**
 - Backups 2, 3: Will prepare m

Splitting replicas

- In general, backups **won't prepare two different requests with the same seqno** if primary lies
- **Suppose they did:** two distinct requests m and m' for the same sequence number n
 - Then prepared quorum certificates (each of size $2f+1$) would **intersect** at an **honest** replica
 - So that honest replica would have sent an accept message for both m and m' which **can't happen**
 - **So $m = m'$**

View change



- If a replica suspects the primary is faulty, it requests a **view change**
 - Sends a **view change** request to all replicas
 - Everyone acks the view change request
- New primary collects a quorum ($2f+1$) of responses
 - Sends a **new-view** message with this certificate

Considerations for view change

- Need committed operations to **survive** into next view
 - Client may have gotten answer
- Need to **preserve liveness**
 - If replicas are too fast to do view change, but really primary is okay – then performance problem
 - Or malicious replica tries to subvert the system by proposing a **bogus view change**

Garbage collection

- Storing all messages and certificates into a **log**
 - Can't let log **grow without bound**

- Protocol to **shrink the log** when it gets too big
 - Discard messages, certificates on commit?
 - No! Need them for view change
 - Replicas have to agree to shrink the log

Proactive recovery

- What we've done so far: good service provided there are no more than f failures **over system lifetime**
 - But cannot **recognize** faulty replicas!
- Therefore **proactive recovery**:
 - **Recover** the replica to a **known good state** whether faulty or not
- Correct service provided no more than f failures in a small time window – e.g., 10 minutes

Recovery protocol sketch

- Watchdog timer
- Secure co-processor
 - Stores node's **private** key (of private-public keypair)
- Read-only memory
- Restart node periodically:
 - Saves its state (timed operation)
 - Reboot, reload code from read-only memory
 - Discard all secret keys (prevent impersonation)
 - Establishes new secret keys and state

Today

1. Traditional state-machine replication for BFT?
2. Practical BFT replication algorithm
[Liskov & Castro, 2001]
- 3. Performance and Discussion**

File system benchmarks

- **BFS** filesystem runs atop BFT
 - Four replicas tolerating one Byzantine failure
 - Modified Andrew filesystem benchmark
- What's performance relative to NFS?
 - Compare BFS versus Linux NFSv2 (unsafe!)
 - **BFS 15% slower: claim can be used in practice**

Practical limitations of BFT

- Protection is achieved only when at most f nodes fail
 - Is one node more or less secure than four?
 - Need **independent implementations** of the service
- Needs **more messages, rounds** than conventional state machine replication
- **Does not prevent** many classes of attacks:
 - Turn a machine into a botnet node
 - Steal data from servers

Large impact

- Inspired **much follow-on work** to address its limitations
- The **ideas surrounding Byzantine fault tolerance** have found numerous applications:
 - Boeing 777 and 787 flight control computer systems
 - Digital currency systems
- Being picked up again in developments of **permissioned blockchain systems**