Byzantine Fault Tolerance

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

So far in COS 418: Fail-stop failures

- Traditional state machine replication
  - Tolerates **fail-stop failures**
    - Node crashes, i.e. stops responding to requests
    - Does not return responses that violate protocol...

- **Result**: State machine replication with $N = 2f + 1$ replicas can tolerate $f$ simultaneous fail-stop failures

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?**
Motivation for BFT

- The ideas surrounding Byzantine fault tolerance have found numerous applications:
  - Commercial airliner flight control computer systems
  - Digital currency systems

- Some limitations, but...
  - Inspired much follow-on research to address these limitations

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 seqno
   - Could assign same seqno to different requests
2. Can't use Paxos for view change
   - Under Byzantine faults, the intersection of two majority \((f + 1 \text{ node})\) quorums may be bad node

   - Bad node tells different quorums different things!
     - e.g. tells N0 accept \text{val1}, but N1 accept \text{val2}
Paxos under Byzantine faults \( (f = 1) \)

- **Prepare** (N0:1)
  - N0
  - N1
  - N2

  \( n_0 = N0:1 \)
  \( n_1 = N0:1 \)

  \( n_2 = N0:1 \)

- **OK** (val = null)
  - N0
  - N1
  - N2

- **Accept** (N0:1, val = xyz)
  - N0
  - N1
  - N2

  \( n_0 = N0:1 \)
  \( n_1 = N0:1 \)
  \( n_2 = N2:1 \)

- **Decide** xyz
  - N0
  - N1
  - N2

- **Decide** abc
  - N0
  - N1
  - N2

- **Conflicting decisions!**
Theoretical fundamentals: Byzantine Generals

Byzantine Generals

Result: Using messengers, problem solvable iff > ⅔ 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 f1="aaa" with server
  – Verify that returned f1 is correctly signed

  But a Byzantine node can replay state, 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 **at least one** reply is 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 \(\geq 1\) 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 replica for sending messages
  - One key per replica for receiving messages

Ordering requests
- 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
- **Let seq(m) = n**
- **I accept seq(m) = n**

Collecting a **prepared certificate** (**f = 1**)
- **Each correct node** is prepared **locally**, but does not **know** whether other correct nodes are prepared! So, **can’t commit** yet!
Collecting a **committed** certificate \((f=1)\)

- **Request:** \(m\)
  - Let \(\text{seq}(m)=n\)
- **Primary**
  - Have cert for \(\text{seq}(m)=n\)
  - Signed, Primary
- **Backup 1**
  - Accept
  - Signed, Backup 1
- **Backup 2**
  - Accept
  - Signed, Backup 2
- **Backup 3**
  - Reject

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 \((f=1)\)

- **Request:** \(m\)
  - Replayed request, signed by client
  - Let \(\text{seq}(m)=n\)
  - Accept \(m\)
- **Primary**
  - Let \(\text{seq}(m)=n\)
- **Backup 1**
  - Let \(\text{seq}(m)=n\)
  - Accept \(m\)
- **Backup 2**
  - Let \(\text{seq}(m)=n\)
  - Accept \(m\)
- **Backup 3**
  - Accept \(m\)

**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 viewchange 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

**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

**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

**Proactive recovery (1)**

- What we’ve done so far: correct service provided ≤ f failures over the system’s lifetime of operation
  - But we can’t recognize faulty replicas!
Proactive recovery (2)

• Therefore proactive recovery:
  – Recover replica to 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 SSNs from servers

Wednesday topic:
Strong consistency and CAP Theorem