Byzantine Fault Tolerance
Fault Tolerance

- Terminology & Background
- Byzantine Fault Tolerance (Lamport)
- Async. BFT (Liskov)
Failure Models

<table>
<thead>
<tr>
<th>Type of failure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Crash failure</td>
<td>A server halts, but is working correctly until it halts</td>
</tr>
<tr>
<td>Omission failure</td>
<td>A server fails to respond to incoming requests</td>
</tr>
<tr>
<td>Receive omission</td>
<td>A server fails to receive incoming messages</td>
</tr>
<tr>
<td>Send omission</td>
<td>A server fails to send messages</td>
</tr>
<tr>
<td>Timing failure</td>
<td>A server’s response lies outside the specified time interval</td>
</tr>
<tr>
<td>Response failure</td>
<td>A server’s response is incorrect</td>
</tr>
<tr>
<td>Value failure</td>
<td>The value of the response is wrong</td>
</tr>
<tr>
<td>State transition failure</td>
<td>The server deviates from the correct flow of control</td>
</tr>
<tr>
<td>Arbitrary failure</td>
<td>A server may produce arbitrary responses at arbitrary times</td>
</tr>
</tbody>
</table>

- A system is said to **fail** if it cannot meet its promises. An **error** on the part of a system’s state may lead to a failure. The cause of an error is called a **fault**.
Process Resilience

- Reaching agreement:
  - computation results
  - Electing a leader
  - synchronization
  - committing to a transaction
  - …

- How much replication is necessary?
  - A system is $k$ fault tolerant if it can survive faults in $k$ components and still meet its specifications.
Agreement in Faulty Systems

- Many things can go wrong…
- Communication
  - Message transmission can be unreliable
  - Time taken to deliver a message is unbounded
  - Adversary can intercept messages
- Processes
  - Can fail or team up to produce wrong results
- Agreement very hard, sometime impossible, to achieve!
Fault Tolerance

- Terminology & Background
- Byzantine Fault Tolerance (Lamport)
- Async. BFT (Liskov)
Byzantine Agreement Problem

System of N processes, where each process $i$ will provide a value $v_i$ to each other. Some number of these processes may be incorrect (or malicious)

Goal: Each process learn the true values sent by each of the correct processes

The Byzantine agreement problem for three nonfaulty and one faulty process.
Byzantine General’s Problem

• The Problem: “Several divisions of the Byzantine army are camped outside an enemy city, each division commanded by its own general. After observing the enemy, they must decide upon a common plan of action. Some of the generals may be traitors, trying to prevent the loyal generals from reaching agreement.”

• Goal:
  • All loyal generals decide upon the same plan of action.
  • A small number of traitors cannot cause the loyal generals to adopt a bad plan.

• The paper considers a slightly different version from the standpoint of one general (i.e. process) and multiple lieutenants.

• Goal:
  • All loyal lieutenants obey the same order.
  • If the commanding general is loyal, the every loyal lieutenant obeys the order he sends.

What we’ve learnt so far: tolerate fail-stop failures

- Traditional replicated state machine (RSM) tolerates benign failures
  - Node crashes
  - Network partitions
- A RSM w/ 2f+1 replicas can tolerate f simultaneous crashes
Why doesn’t traditional RSM work with Byzantine nodes?

• Cannot rely on the primary to assign seqno
  • Malicious primary can assign the same seqno to different requests!

• Cannot use Paxos for view change
  • Paxos uses a majority accept-quorum to tolerate f benign faults out of 2f+1 nodes
  • Does the intersection of two quorums always contain one honest node?
  • Bad node tells different things to different quorums!
    • E.g. tell N1 accept=val1 and tell N2 accept=val2
Paxos under Byzantine faults

- Prepare vid=1, myn=N0:1
  - OK val=null

Diagram:
- N2
  - nh=N0:1

- N0
  - nh=N0:1
  - Prepare vid=1, myn=N0:1
    - OK val=null

- N1
  - nh=N0:1
Paxos under Byzantine faults

accept vid=1, myn=N0:1, val=xyz
OK

N0 decides on Vid1=xyz

N0

n\text{h}=N0:1

N2

n\text{h}=N0:1

N1

N0 and N1 are Byzantine faults.

N0 decides on Vid1=xyz.
Paxos under Byzantine faults

prepare vid=1, myn=N1:1, val=abc
OK val=null

N0 decides on Vid1=xyz

N0
nh=N0:1

N2

N1
nh=N0:1
Paxos under Byzantine faults

N0 decides on Vid1=xyz

N1 decides on Vid1=abc

Agreement conflict!
BFT requires a 2f+1 quorum out of 3f+1 nodes

For liveness, the quorum size must be at most N - f
BFT Quorums

For correctness, any two quorums must intersect at least one honest node: \((N-f) + (N-f) - N \geq f+1 \rightarrow N \geq 3f+1\)
Agreement in Faulty Systems

- Possible characteristics of the underlying system:
  1. Synchronous versus asynchronous systems.
     - A system is synchronized if the process operation in lock-step mode. Otherwise, it is asynchronous.
  2. Communication delay is bounded or not.
  3. Message delivery is ordered or not.
  4. Message transmission is done through unicasting or multicasting.
Agreement in Faulty Systems

- Circumstances under which distributed agreement can be reached. Note that most distributed systems assume that:
  1. processes behave asynchronously
  2. messages are unicast
  3. communication delays are unbounded (see red blocks)
Synchronous, Byzantine world

- Synchronous
- Asynchronous
- Fail-stop
- Byzantine
Agreement in Faulty Systems - 4

- Byzantine Agreement [Lamport, Shostak, Pease, 1982]
- Assumptions:
  - Every message that is sent is delivered correctly
  - The receiver knows who sent the message
  - Message delivery time is bounded

![Message ordering table](image)
Byzantine Agreement Algorithm (oral messages) - 1

- Phase 1: Each process sends its value to the other processes. Correct processes send the same (correct) value to all. Faulty processes may send different values to each if desired (or no message).

- Assumptions: 1) Every message that is sent is delivered correctly; 2) The receiver of a message knows who sent it; 3) The absence of a message can be detected.

Byzantine General Problem
Example - 1

- Phase 1: Generals announce their troop strengths to each other
Byzantine General Problem
Example - 2

- Phase 1: Generals announce their troop strengths to each other

Diagram:

- P1
- P2
- P3
- P4

Connections:
- P1 to P2: 2
- P2 to P3: 2
- P2 to P4: 2
- P3 to P1: 2
Byzantine General Problem
Example - 3

- Phase 1: Generals announce their troop strengths to each other
Byzantine Agreement Algorithm (oral messages) - 2

- Phase 2: Each process uses the messages to create a vector of responses – must be a default value for missing messages.

- Assumptions: 1) Every message that is sent is delivered correctly; 2) The receiver of a message knows who sent it; 3) The absence of a message can be detected.

Byzantine General Problem Example - 4

- Phase 2: Each general construct a vector with all troops

<table>
<thead>
<tr>
<th></th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>1</td>
<td>2</td>
<td>x</td>
<td>4</td>
</tr>
</tbody>
</table>

P1 → P3: x
P2 → P3: y
P3 → P4: z
P4 → P1: 4
P4 → P2: 4
Byzantine Agreement Algorithm (oral messages) - 3

- Phase 3: Each process sends its vector to all other processes.
- Phase 4: Each process the information received from every other process to do its computation.

Assumptions: 1) Every message that is sent is delivered correctly; 2) The receiver of a message knows who sent it; 3) The absence of a message can be detected.

Byzantine General Problem
Example - 5

- Phase 3,4: Generals send their vectors to each other and compute majority voting
Byzantine Agreement Algorithm (signed messages)

- Adds the additional assumptions:
  - A loyal general’s signature cannot be forged and any alteration of the contents of the signed message can be detected.
  - Anyone can verify the authenticity of a general’s signature.

- Algorithm SM(m):
  - The general signs and sends his value to every lieutenant.
  - For each i:
    - If lieutenant i receives a message of the form v:0 from the commander and he has not received any order, then he lets Vi equal {v} and he sends v:0:i to every other lieutenant.
    - If lieutenant i receives a message of the form v:0:j1:…:jk and v is not in the set Vi then he adds v to Vi and if k < m, he sends the message v:0:j1:…:jk:i to every other lieutenant other than j1,…,jk
    - For each i: When lieutenant i will receive no more messages, he obeys the order in choice(Vi).

- Algorithm SM(m) solves the Byzantine General’s problem if there are at most m traitors.

Signed messages

SM(1) with one traitor

Fault Tolerance

- Terminology & Background
- Byzantine Fault Tolerance (Lamport)
- Async. BFT (Liskov)
Practical Byzantine Fault Tolerance: Asynchronous, Byzantine
Practical Byzantine Fault Tolerance

• Why async BFT? BFT:
  • Malicious attacks, software errors
  • Need N-version programming?
  • Faulty client can write garbage data, but can’t make system inconsistent (violate operational semantics)

• Why async?
  • Faulty network can violate timing assumptions
  • But can also prevent liveness
Distributed systems

- FLP impossibility: Async consensus may not terminate
  - **Sketch of proof:** System starts in “bivalent” state (may decide 0 or 1). At some point, the system is one message away from deciding on 0 or 1. If that message is delayed, another message may move the system away from deciding.
  - Holds even when servers can only crash (not Byzantine)!
  - Hence, protocol cannot always be live (but there exist randomized BFT variants that are probably live)


In the system Fischer, Lynch, and Paterson studied, messages were unordered, communication was unbounded, and processors were asynchronous.
PBFT ideas

- PBFT, “Practical Byzantine Fault Tolerance”, M. Castro and B. Liskov, SOSP 1999
- Replicate service across many nodes
  - Assumption: only a small fraction of nodes are Byzantine
  - Rely on a super-majority of votes to decide on correct computation.
  - Makes some weak synchrony (message delay) assumptions to ensure liveness
    - Would violate FLP impossibility otherwise
- PBFT property: tolerates $\leq f$ failures using a RSM with 3f+1 replicas
PBFT main ideas

- Static configuration (same 3f+1 nodes)
- To deal with malicious primary
  - Use a 3-phase protocol to agree on sequence number
- To deal with loss of agreement
  - Use a bigger quorum (2f+1 out of 3f+1 nodes)
- Need to authenticate communications
PBFT Strategy

• Primary runs the protocol in the normal case
• Replicas watch the primary and do a view change if it fails
Replica state

- A replica id \( i \) (between 0 and \( N-1 \))
  - Replica 0, replica 1, …
- A view number \( v\# \), initially 0
- Primary is the replica with id
  \[ i = v\# \mod N \]
- A log of \( <\text{op}, \text{seq}\#, \text{status}> \) entries
  - Status = pre-prepared or prepared or committed
Normal Case

- Primary sends **pre-prepare** message to all
- Pre-prepare contains $<v#,seq#,op>$
  - Records operation in log as pre-prepared

- Keep in mind that primary might be malicious
  - Send different seq# for the same op to different replicas
  - Use a duplicate seq# for op
PBFT

Client

Primary

Replica 2

Replica 3

Replica 4

<table>
<thead>
<tr>
<th>Request</th>
<th>Pre-Prepare</th>
<th>Prepare</th>
<th>Commit</th>
<th>Reply</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Normal Case

- Replicas check the pre-prepare and if it is ok:
  - Record operation in log as pre-prepared
  - Send prepare messages to all
  - Prepare contains \( <i,v#\),seq#,op> \)

- All to all communication
PBFT

- **Request**
- **Pre-Prepare**
- **Prepare**
- **Commit**
- **Reply**

Roles:
- **Client**
- **Primary**
- **Replica 2**
- **Replica 3**
- **Replica 4**
Normal Case:

- Replicas wait for \(2f+1\) matching prepares
  - Record operation in log as prepared
  - Send commit message to all
  - Commit contains \(<i,v\#,seq\#,op>\)
- What does this stage achieve:
  - All honest nodes that are prepared prepare the same value
  - At least \(f+1\) honest nodes have sent prepare/pre-prepare
PBFT

<table>
<thead>
<tr>
<th>Request</th>
<th>Pre-Prepare</th>
<th>Prepare</th>
<th>Commit</th>
<th>Reply</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Primary</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Replica 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Replica 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Replica 4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Normal Case:

- Replicas wait for $2f+1$ matching commits
  - Ensures that at least $f+1$ trustworthy nodes have committed
- Record operation in log as committed
  - Execute the operation
  - Send result to the client
PBFT

Client

Primary

Replica 2

Replica 3

Replica 4

Request | Pre-Prepare | Prepare | Commit | Reply
Normal Case

- Client waits for $f+1$ matching replies
  - Ensures at least one node has committed and executed
View Change

- Replicas watch the primary
- Request a view change
  - send a do-viewchange request to all
  - new primary requires $2f+1$ requests to accept new role
  - sends new-view with proof that it got the previous messages
Practical limitations of BFTs

• Expensive
• Protection is achieved only when \( \leq f \) nodes fail
• Does not prevent many types of attacks:
  • Turn a machine into a botnet node
  • Steal SSNs from servers