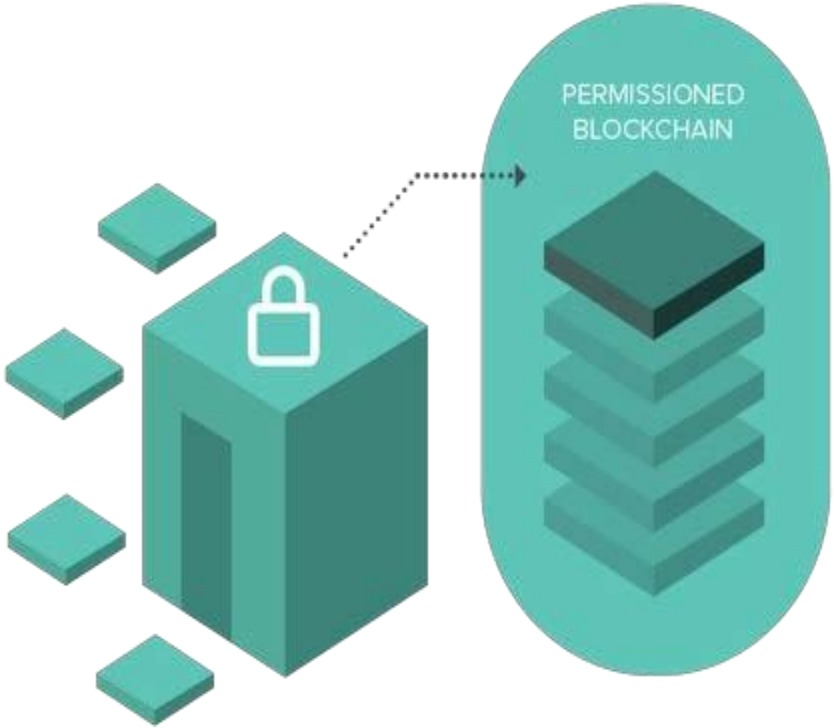


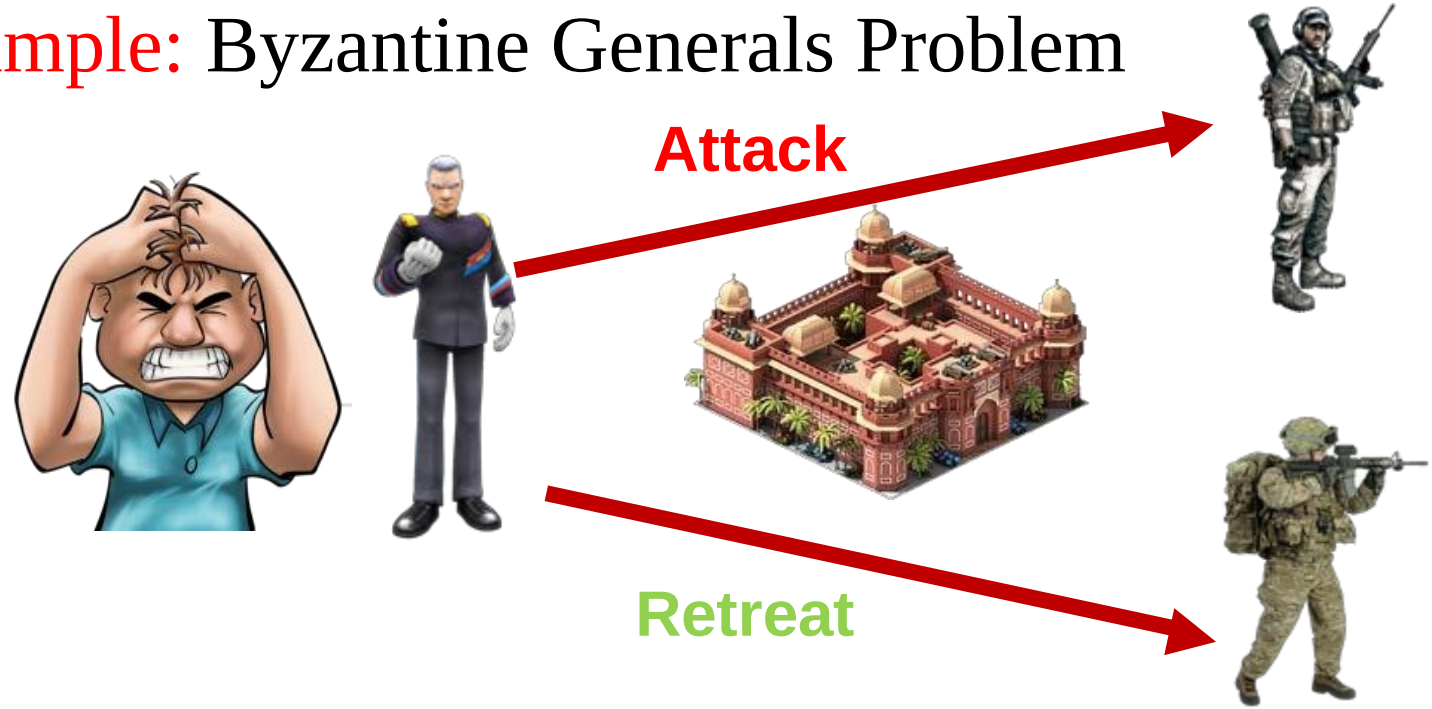
Image Source: https://nem.io/enterprise/

Permissioned Blockchain - IV
Consensus Algorithms

- We have already talked about the fault tolerance nature of distributed consensus protocols
- Already discussed about **Paxos and RAFT** consensus protocols.
- Both works good to handle the Crash faults
- But if there is **some byzantine behavior** in the n/w where nodes behaving maliciously then both protocols can’t handle this kind of scenario
- So there is a need of different class of fault tolerance protocols for distributed system under the permissioned or the closed environment.
- These are known as “**Byzantine fault tolerance protocols”** ??

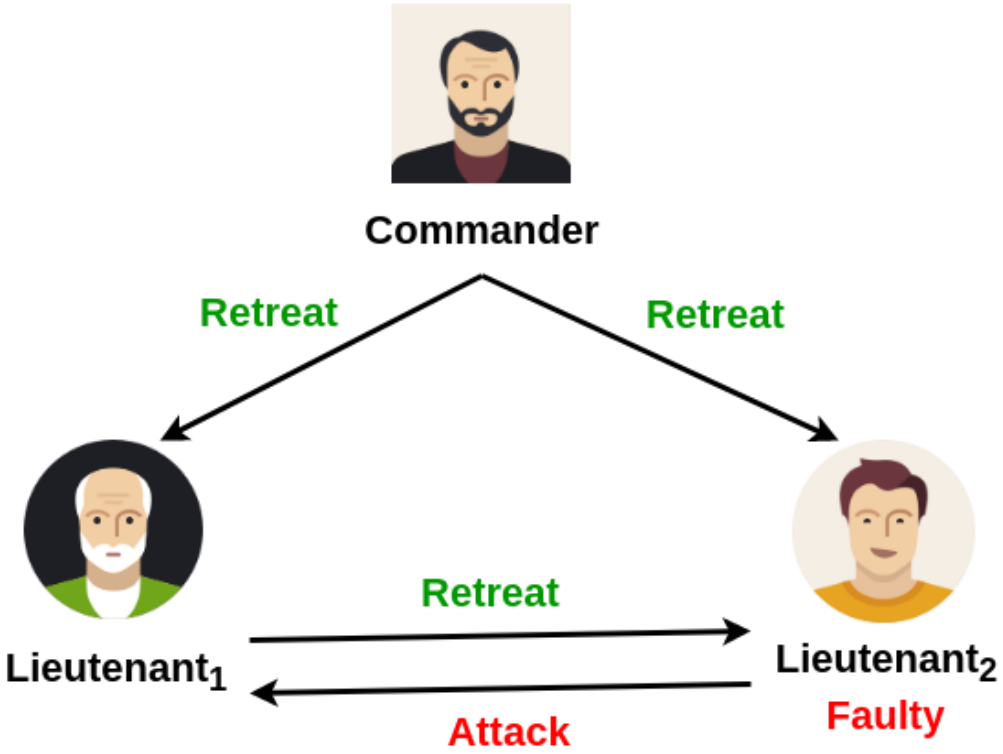
- In the permissioned model, fault can occur, which can be categorized in to **three categories**:
- **Crash Fault** where a node can fail arbitrarily due to crash that can be a hardware or software crash
- In case of crash fault, it may happen that node stops transmitting messages to all its peers. But crash faults can be recoverable so a node can be recovered from crash fault after certain duration and after this recovery, nodes can start behaving normally.
- Crash fault has another variants; n/w failure where the entire n/w is partitioned in to two n/w's so there is no communication between the nodes of both the partitioned n/w's.

Example: Byzantine Generals Problem



- To handle crash fault, paxos and Raft can be used to frame out the consensus.
- But in case of Byzantine fault, it may happen that node may sends different messages to different peers.
- If the General becomes faulty then it is difficult for the system to find out what to do?

Three Byzantine Generals Problem under multiple node scenario: **Lieutenant Faulty**

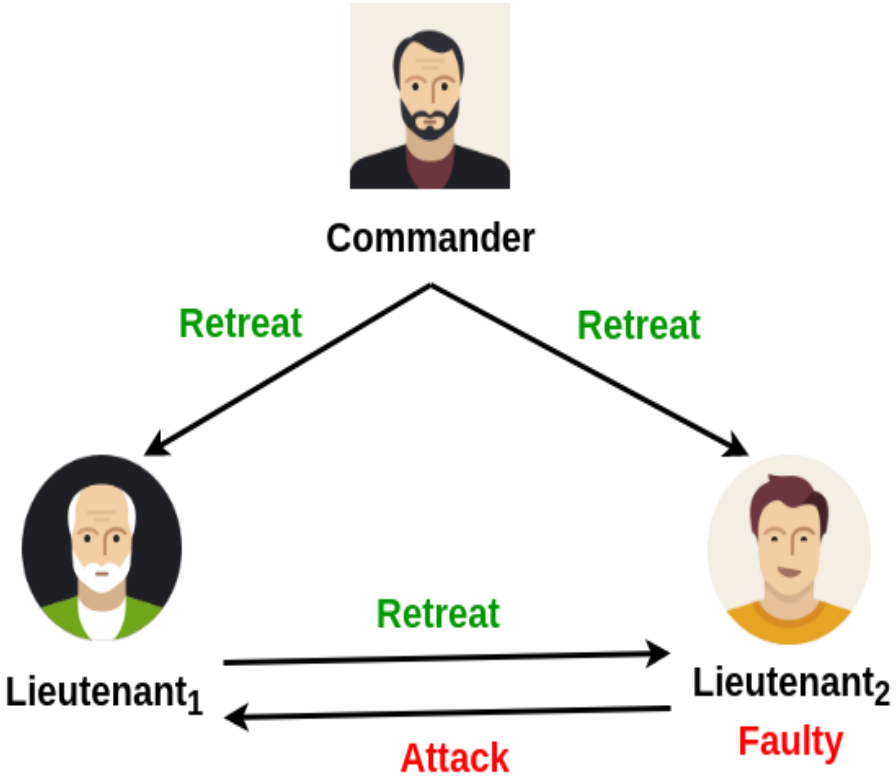


- Round1:
  - Commander correctly sends same message to Lieutenants
- Round 2:
  - Lieutenant_1 correctly echoes to Lieutenant_2
  - Lieutenant_2 **incorrectly** echoes to Lieutenant_1

Here, we assume that Liettenant2 is faulty.

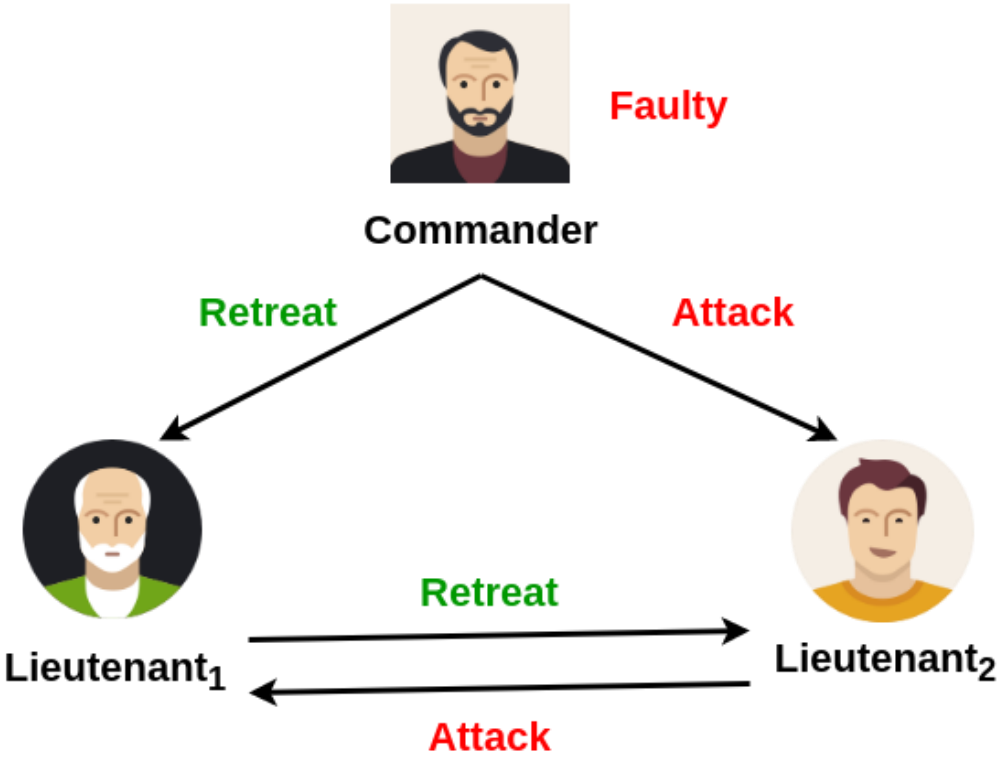
Now here commander is correct and one of the Lt. is faulty so how can, we frame out the consensus ??

Three Byzantine Generals Problem: Lieutenant Faulty



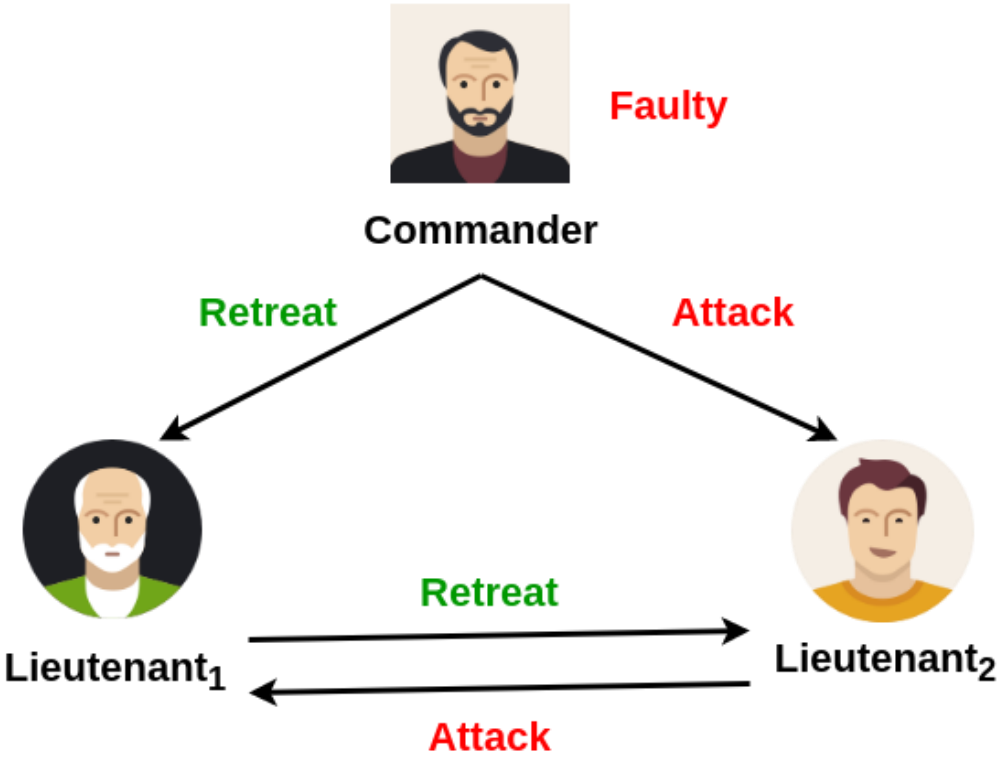
- Here, Lieutenant$_1$ received **differing message**
  In normal military scenario, the Lt. obey the commander order, if the **commander is not faulty** then the entire system works perfectly
- By integrity condition, Lieutenant$_1$ bound to decide on Commander message
- **What if Commander is faulty??**

Three Byzantine Generals Problem: Commander Faulty



- Round 1:
  - Commander sends **differing message** to Lieutenants
- Round 2:
  - $Lieutenant_1$ correctly echoes to $Lieutenant_2$
  - $Lieutenant_2$ correctly echoes to $Lieutenant_1$
- Here, we assume that Commander is faulty and both the Lts are correct
- By **message passing**. the entire system will not be able to work

Three Byzantine Generals Problem: Commander Faulty

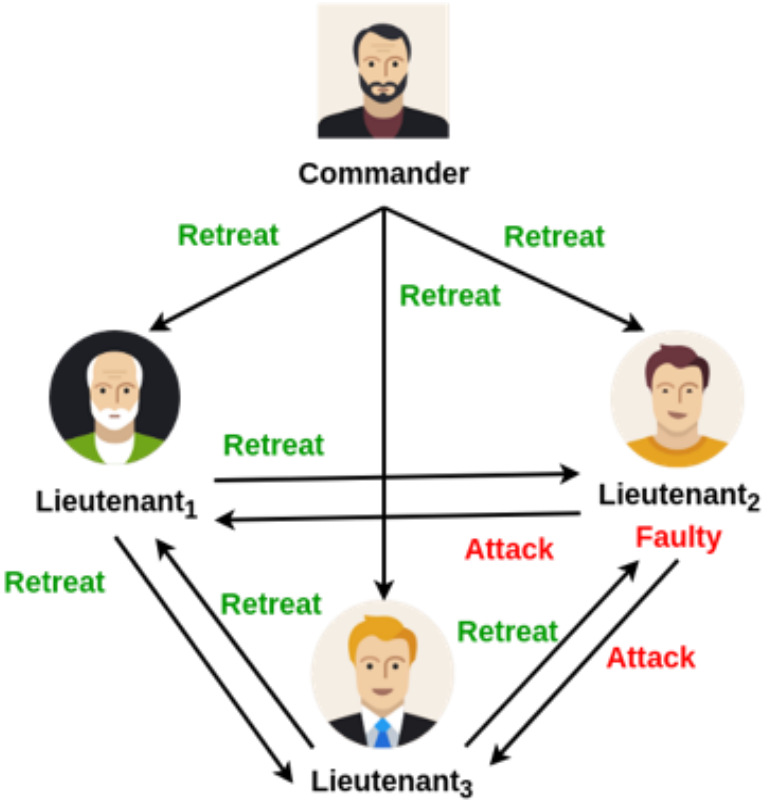


- Lieutenant$_1$received **differing message**
- By integrity condition, both Lieutenants conclude with Commander's message
- This contradicts the agreement condition
- **No solution** possible for three generals including one faulty

Three Byzantine Generals Problem: Commander Faulty

- If we don't have any way **to finalize whether to go for majority voting or to follow commander instructions** then the entire system in a **dilemma** that which particular instructions is to be followed.
- We can solve this byzantine problem with the principle of **majority voting** (we have seen in the case of paxos or Raft)
- In a byzantine system, if the leader (commander) itself sends different messages to different peers (Lts) then coming up to consensus on this principle is very difficult.
- So, we will see through one example how the followers (Lts) in the Raft terminology, various followers (Lts) starts sending messages to each other and they can look through majority principle (by voting against or in favor of the leader (commander)) that whether the leader is faulty or not.
- But the three Generals problem with one commander and two Lts, the problem is still un solvable through voting principle.
- Another use case with four Generals

Four Byzantine Generals Problem: **Lieutenant Faulty**



All Lts talks with each other by message passing principle

Round 1:

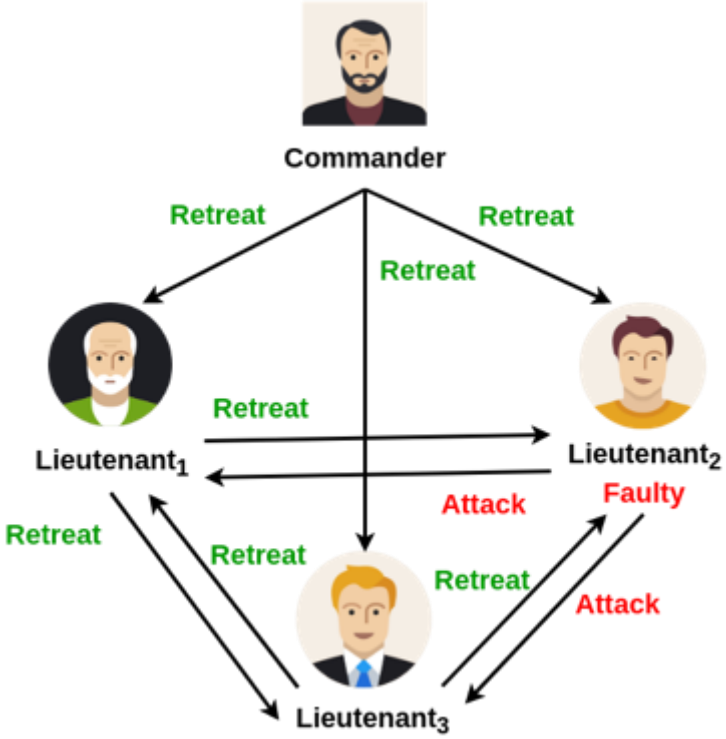
Commander sends a message to each of the Lieutenants

Round 2:

$Lieutenant_1$ and $Lieutenant_3$ correctly echo the message to others

$Lieutenant_2$ **(faulty) incorrectly** echoes to others

Four Byzantine Generals Problem: **Lieutenant Faulty**



In this scenario, if we go with majority principle then

- **Lieutenant$_1$** decides on *majority*(Retreat,Attack,Retreat)= **Retreat**
- **Lieutenant$_3$** decides on *majority*(Retreat,Retreat,Attack)= **Retreat**
- **Lieutenant$_2$ is faulty** so we have not considered his behavior

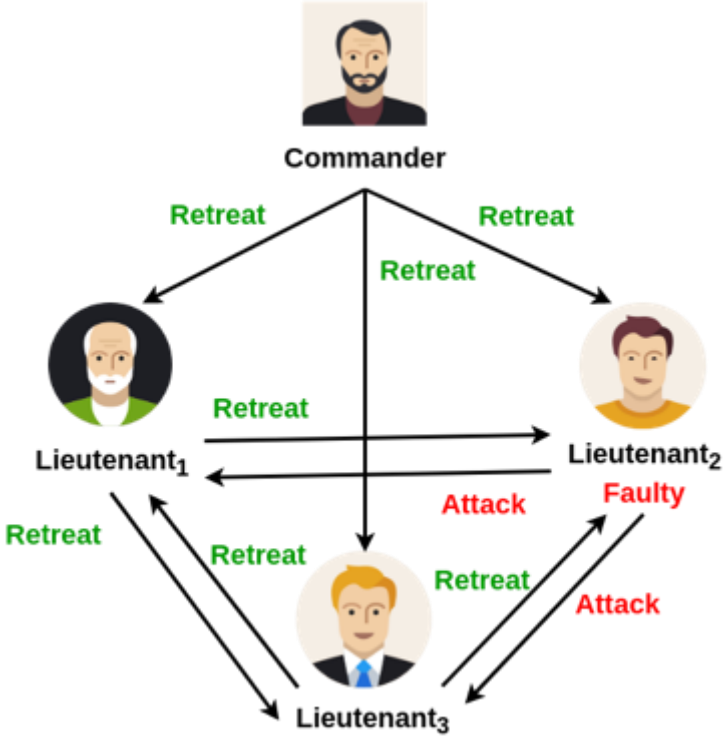
Observation:

- Objective of fault tolerance algorithm means the integrity principle; all the correct nodes in the fault tolerance system will follow the majority vote.

- Even if Lt 2is faulty, the Lt1 and Lt2 decode the message correctly by majority voting principle

What happened if two Lts are faulty??

Four Byzantine Generals Problem: **Two Lieutenant Faulty**



In this scenario, if Lt1 and Lt 2 are faulty then rather than sending correct message (Retreat) Lt1 sends Attack

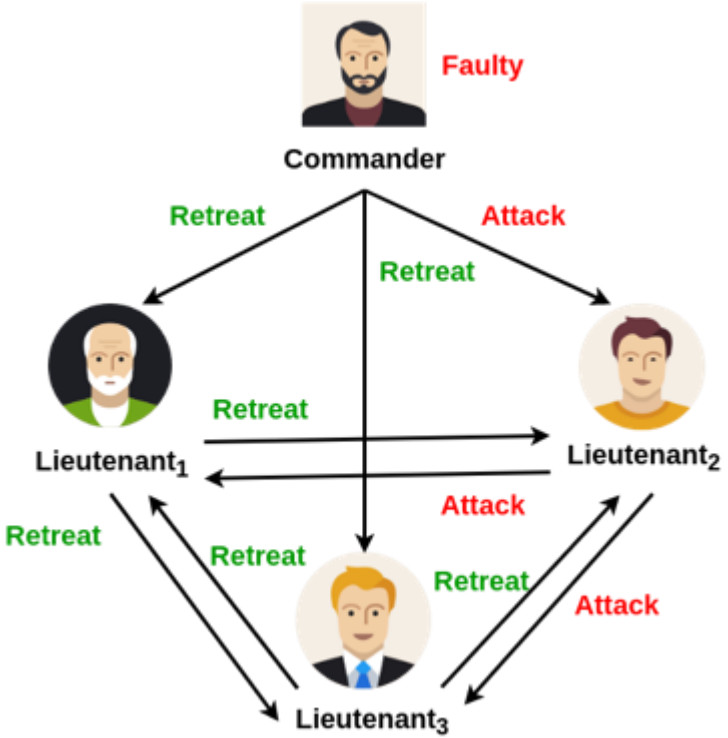
- **Lieutenant$_3$** decides on *majority*(Retreat, Attack, Attack)= Attack so no correct decision based on majority voting

Observation:

- Out of the 3 Lts if one Lt is faulty then we will be able to correctly decode the message
- But if out of the 3 Lts 2 Lts are faulty then we will not be able to correctly decode the message

What happened if the commander is faulty??

Four Byzantine Generals Problem: **Commander Faulty**



Means the commander starting behaving maliciously

Round 1:

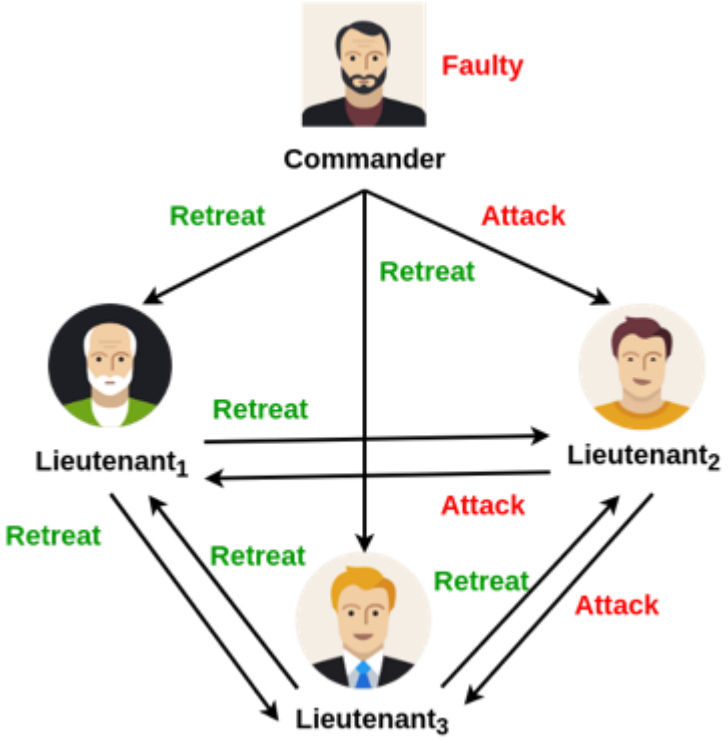
Commander sends differing message to Lieutenants

Round 2:

$Lieutenant_1$, $Lieutenant_2$ and $Lieutenant_3$ correctly echo the message to others (means no Lts are faulty)

Let us see majority voting principle works here or not?

Four Byzantine Generals Problem: **Commander Faulty**



- Lieutenant$_1$ decides on *majority*(Retreat,Attack,Retreat)= **Retreat**
- Lieutenant$_2$ decides on *majority*(Attack,Retreat,Retreat)= **Retreat** (This decision identifies that Commander is faulty)
- Lieutenant$_3$ decides on *majority*(Retreat,Retreat,Attack)= **Retreat**

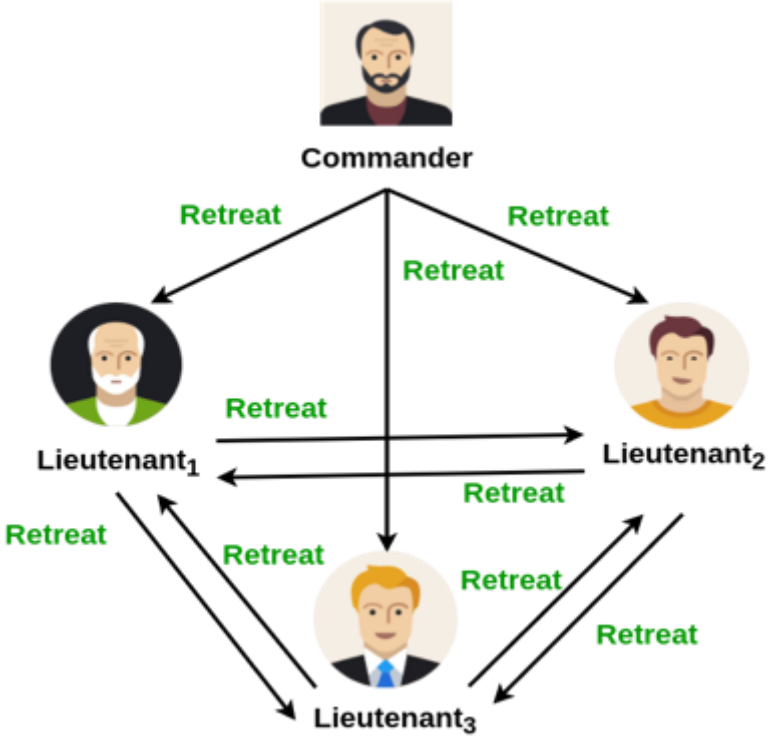
Observation:

If there are three different Lts who are correct and one commander (behaving maliciously) then majority principle will give you correct result.

How to find out Byzantine node in the Four Byzantine Generals Problem?

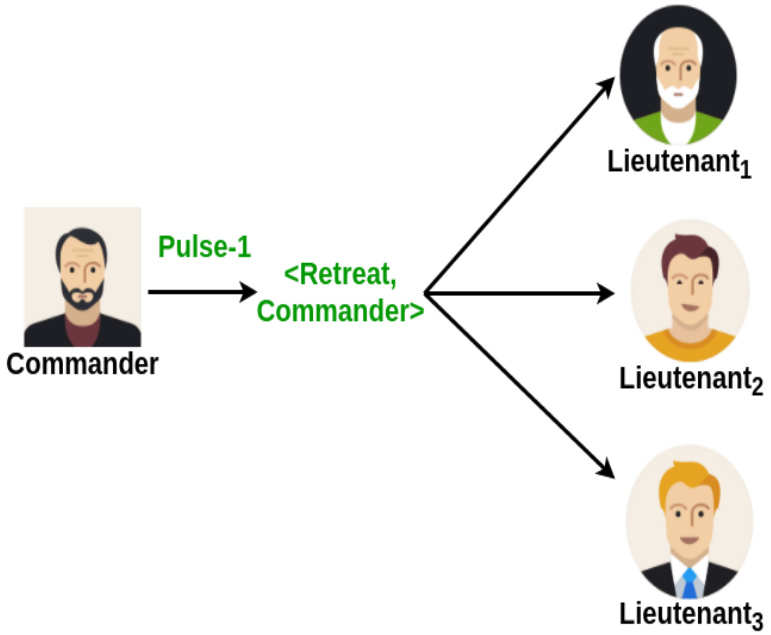
- If we look in to the normal byzantine general problem with *f* number of faulty nodes
- then we can say that if the Lt is faulty then we need to ensure that there are 2f+1 number of Lts in the system
- means if we have (2f+1) Lts + 1 commander
- then we can correctly apply this majority principle to find out the byzantine node in the system
- If the commander is faulty then the system will come to the consensus with a value that was sent by the commander to majority of the Lts

Byzantine Generals Model: General description



- *N* number of process with at most *f* Faulty (ensure that there are 2f+1 Lts in the system)
- Receiver always knows the identity of the sender (means closed model)
- Fully connected
- Reliable communication medium
- Synchronous system (means every node will be able to receive all the messages within some predefined time duration)

Lamport-Shostak-Pease Algorithm (Syn)



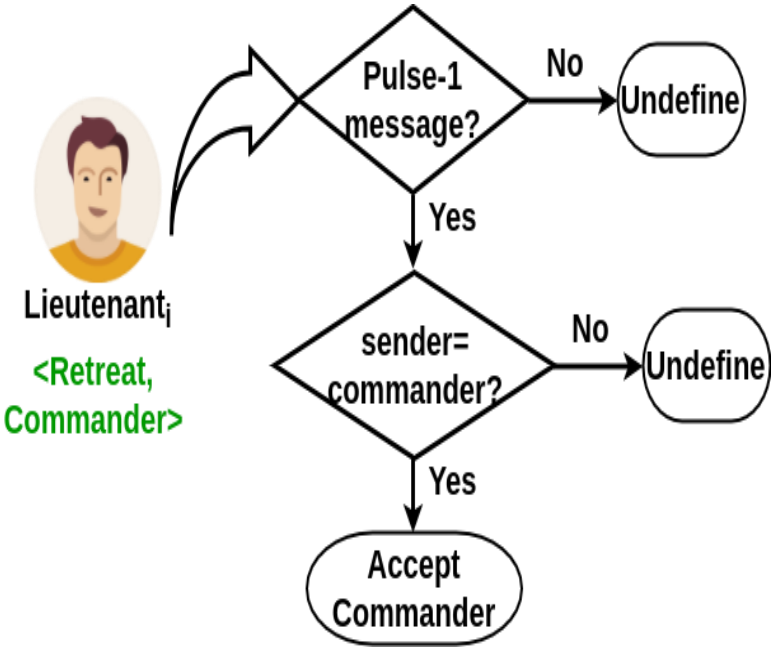
- **Base Condition for the commander contains two parameters:**

  Broadcast(N, t=0)

  - N: number of processes
  - t: algorithm parameter (denotes the individual rounds)
  - E.g) t=0 means you are in pulse 0 when the commanders sends the massage to all Lts
  - N=3 because we have three Lts
- Commander decides on its own value (whether to go for retreat or attack)
- First Algorithm to discuss the Byzantine General problem

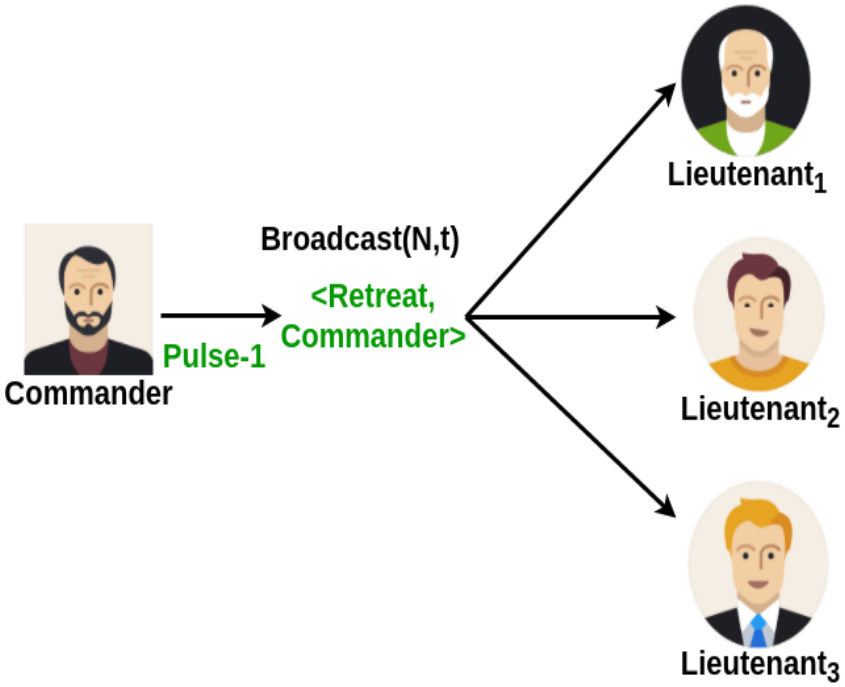
Source: Lamport, Leslie, Robert Shostak, and Marshall Pease. "The Byzantine generals problem." ACM Transactions on Programming Languages and Systems (TOPLAS) 4.3 (1982): 382-401.

Lamport-Shostak-Pease Algorithm



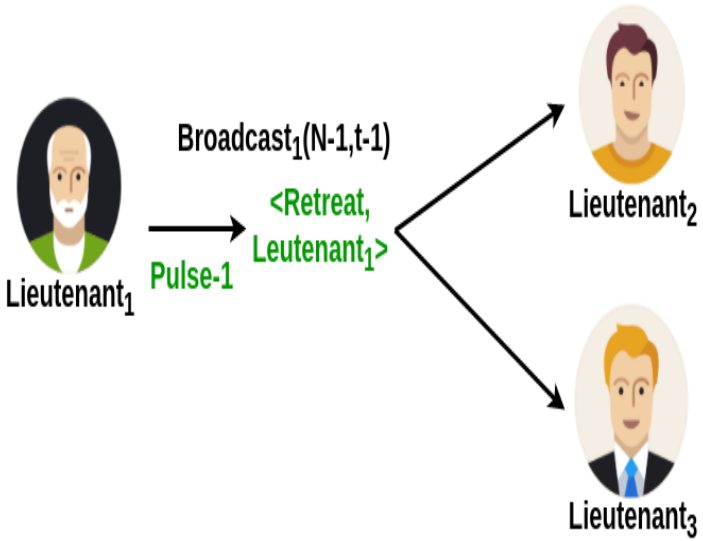
- **Base Condition for Lt.:**
  Broadcast(N,t=0)
  - N: number of processes
  - t: algorithm parameter
- Lieutenants decision by sender matching
- Lt receives the message from the commander then check whether its is coming from commander (pulse-1(first message in the system)) if no then don't take any decision because source is not genuine.
- Otherwise, accept the commander message

Lamport-Shostak-Pease Algorithm



- **General Condition:**
  Broadcast(N,t)
  - N: number of processes
    t: algorithm parameter (round)
- Only commander sends to all lieutenants
- Now broadcast the received message to all other processes in the system.
- As system progresses in rounds so at every individual round, say commander T send the message to all in the Tth round.

Lamport-Shostak-Pease Algorithm for Sync environment



- **General Condition:**
  Broadcast(N,t)
  - N: number of processes
  - t: algorithm parameter
- All lieutenants broadcast their values to the other lieutenants except the senders
- Similarly for the every individual Lts, what ever messages they broadcast the message that they have received from commander to all other Lts except the sender of that particular message
- If you have *N* numbers of Lts in the system then after Nth round you are getting the messages from all the individual Lts
- Then, you can apply the majority principle to take decision (whether to follow the commander or majority of Lts)

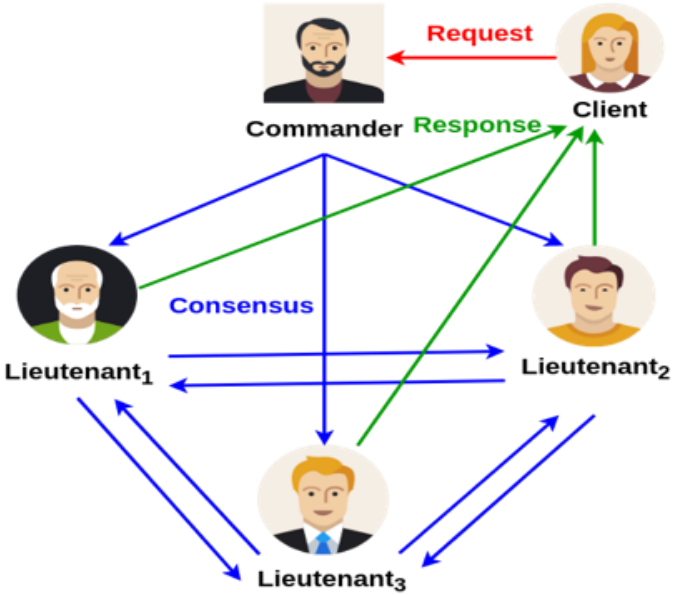
Outcome

- But our practical networking systems are asynchronous where it may happened that you may not receive message with in a predefined timeout
- In next part, we will look in to another set of consensus problem which provides safety guaranty for asynchronous n/w but there is certain conditions on the liveness principle, we can call it as **Practical Byzantine Fault Tolerant (PBFT)**

Practical Byzantine Fault Tolerant (PBFT)

- Why **Practical**? Because:
  - Ensures safety over an asynchronous network (not liveness! Otherwise it violates the impossibility theorem)
  - Byzantine Failure
  - Low overhead
- **Widely applicable for many real Applications**
  - Tendermint
  - IBM's Openchain
  - ErisDB
  - Hyperledger

Practical Byzantine Fault Tolerant (PBFT) Model



- Here the **client** (individual BC clients who are submitting the TXs) submits the request to the **commander** (nodes in the system who are responsible for ensuring consensus in the system).
- It is Asynchronous distributed system
  - **Delay** in transmitting messages, out of order messages
- Byzantine failure handling
  - arbitrary node behavior (means some nodes can vote in favor or against a Tx)
- It supports Privacy over the system
  - It ensures that the message is **tamper-proof** (Using std hashing techniques similar to BC), also **apply authentication** techniques like digital signature so that none of the messages transferred from individual nodes in the system can be tampered or eavesdropped

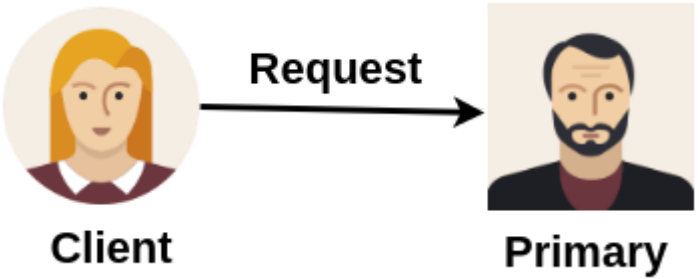
Practical Byzantine Fault Tolerant System Model

- A state machine is replicated across different nodes
- $3f + 1$ replicas are there where f is the number of faulty replicas
- The replicas move through a successions of configurations, known as ***views***
- One replica in a *view* is ***primary*** and others are ***backups***

Practical Byzantine Fault Tolerant Model

- Views are changed when a ***primary*** is detected as faulty
- Every view is identified by a **unique integer number *v***
- **Only the messages from the current views are accepted**, means if there is view change then the messages which have been broadcasted in the previous view are discarded (as in async mode may be you are getting delayed and out of order messages)

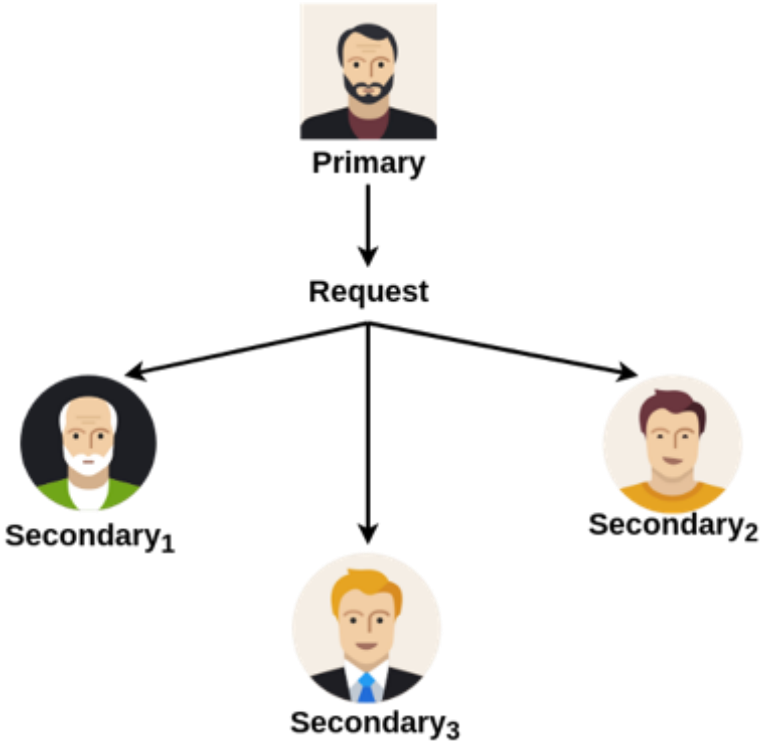
Practical Byzantine Fault Tolerant Algorithm-Execution



- **Idea:** A client sends a request to invoke a service operation to the primary
- Service operation like in case of your share storage, the client executes the write instructions or in case of a BC environment when a client initiates a Tx
- All these client requests moved to the primary

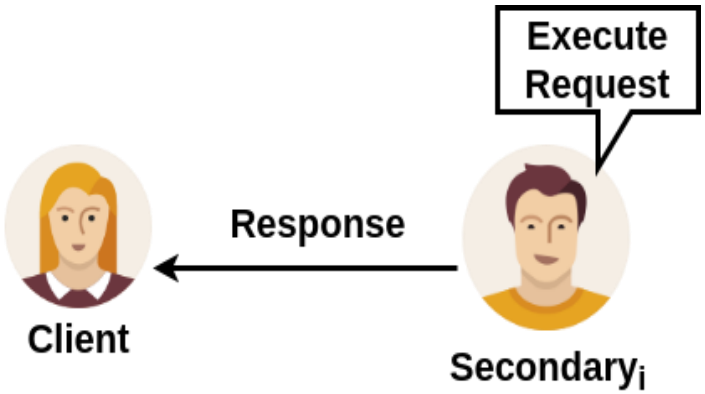
Castro, Miguel, and Barbara Liskov. "Practical Byzantine fault tolerance." OSDI. Vol. 99. 1999.

Practical Byzantine Fault Tolerant Algorithm



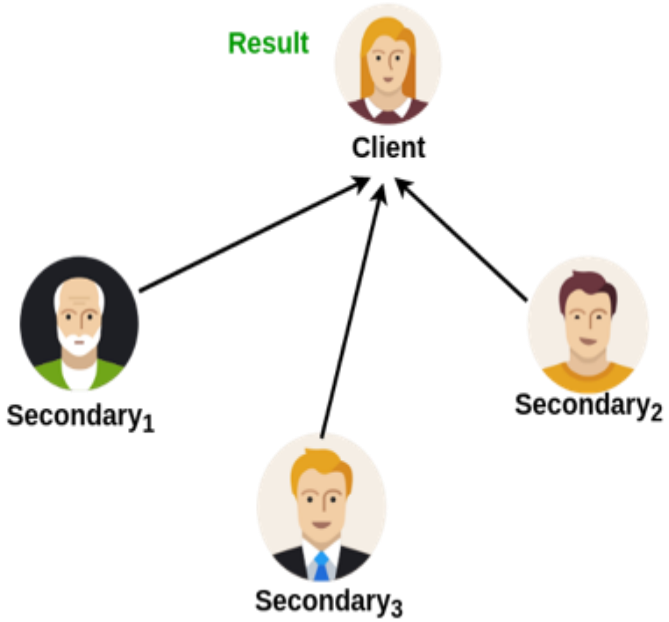
- The primary multicasts the request to the **backups (means secondary replicas)**

Practical Byzantine Fault Tolerant Algorithm



- Backups execute the request and they come to consensus based on PBFT algorithm send a reply to the client
- Client request can be successful or failure
- **For example,** In case of a share storage architecture, if the client is trying to initiate a disk write operation then either this operation successful or failure
- **Another Example**, In BC, if it is a valid Tx if all the backups or replicas decide that the current TX is a correct TX then it send the success or commit message to the client otherwise send the failure message to the client

Practical Byzantine Fault Tolerant Algorithm

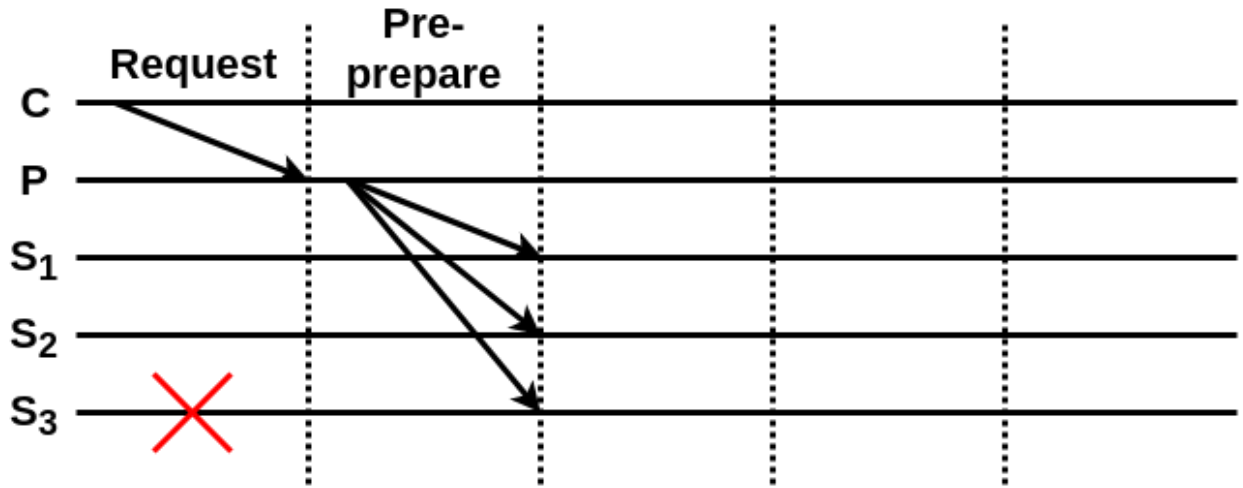


- The client waits for $f + 1$ replies from different backups with the same result
  - f is the maximum number of faulty replicas that can be tolerated
  - Result same means the client can decides that it has got majority of the correct voting
- Remember in PBFT you gave 3f+1 number of different replicas so out of these 3f+1 replicas, you have f numbers of replicas which are faulty but whenever client is receiving the messages
- then majority of the 2f+1 correct nodes sending the reply to you then you can say that system reached to the consensus and
- the client accept or commit that particular message

PBFT executes in Three Phase Commit Protocol - **Pre-Prepare**

- **In Pre-prepare phase:** Primary assigns a sequence number n to the request and multicast a message $<< PRE-PREPARE, v, n, d >_{\sigma\_p}, m >$ to all the backups
  - v is the current view number (to ensure you are getting the message from current view)
  - n is the message sequence number
  - d is the message digest
  - σ_p is the private key of primary - works as a digital signature (Means this entire pre-prepare message: $<< PRE-PREPARE, v, n, d >$ is encrypted)
  - m is the message to transmit

Three Phase Protocol



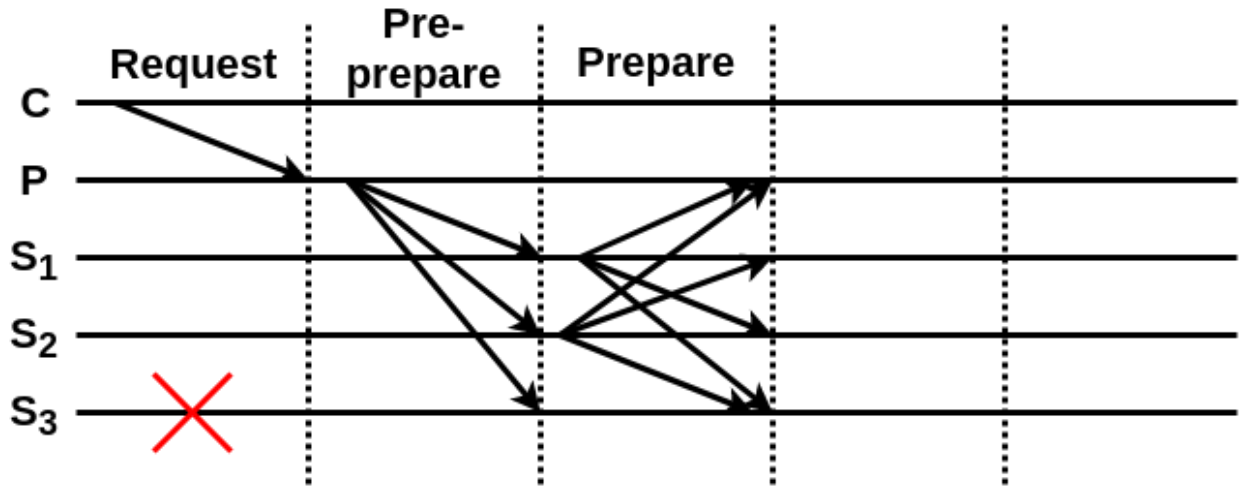
Here, Secondary-S3 is faulty means 3f+1 number of different replicas, if f=1 the we have four replicas one Primary (P) and three secondary S1, S2, and S3.

- **Pre-prepare:**
  - Acknowledge the request by a unique sequence number

Three Phase Commit Protocol - Pre-Prepare

- Pre-prepare messages are used as a proof that request was assigned sequence number n is the view v
- A backup accepts a pre-prepare message if
  - The signature is correct and d is the digest for m
  - The backup is in view v
  - It has not received a different PRE-PREPARE message with sequence n and view v with a different digest (Means message is not tampered)
  - The sequence number is within a threshold (Means sequence number is not too old, which open the doors for the replay attack)

Second Phase is **Prepare Phase** in Three Phase Protocol



- Prepare:
  - Replicas agree on the assigned sequence number

Three Phase Commit Protocol - Prepare

- If the backup accepts the PRE-PREPARE message, it enters prepare phase by multicasting a message < *P*REPARE, v, n, d, i $>_{\sigma\_i}$ to all other replicas

- A replica (both primary and backups) accepts prepare messages if
  - Signatures are correct (**Means coming from the indented backups**)
  - View number equals to the current view **(Means the message is in the current view)**
  - Sequence number is within a threshold **(Means sequence number is not too old, which open the doors for the replay attack)**

Three Phase Commit Protocol

- **Pre-prepare and prepare** ensure that non-faulty replicas guarantee on a total order for the requests within a view.
- Commit a message if
  - $2f$ prepares from different backups matches with the corresponding pre-prepare
  - You have total $2f + 1$ votes **(one from primary that you already have!) from the non-faulty replicas**

Three Phase Commit Protocol

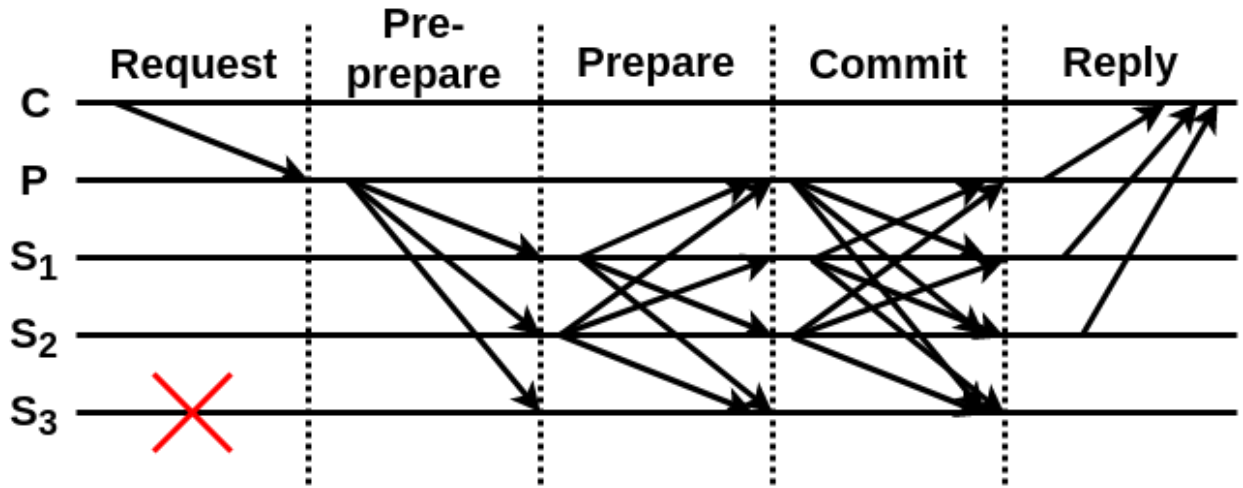
- **Why do you require 3f + 1 replicas to ensure safety in an asynchronous system when there are f faulty nodes?**
  - If you have $2f + 1$ replicas, you need all the votes to decide the majority - boils down to a synchronous system
  - You may not receive votes from certain replicas due to delay, in case of an asynchronous system
  - If you receives $f + 1$ votes that do not ensure majority, may be you have received f votes from Byzantine nodes and just one vote from a non-faulty node (note Byzantine nodes can vote for or against - You do not know that apriori!)

Three Phase Commit Protocol - **Commit**

- Multicast $< COMMIT, v, n, d, i >_{\sigma\_i}$ message to all the replicas including primary

- Commit a message when a replica
  - Has sent a commit message itself
  - Has received $2f + 1$ commits (including its own)

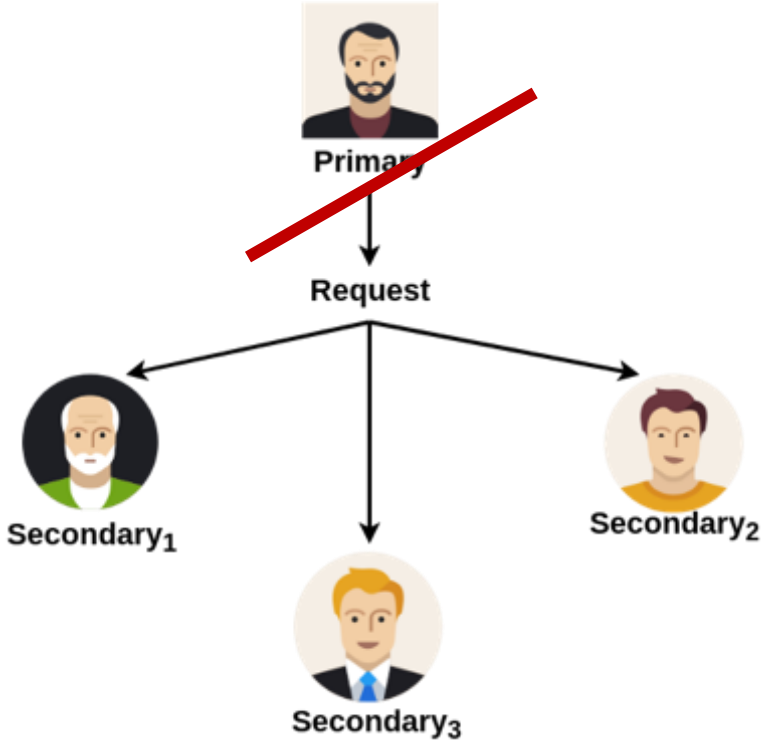
Three Phase Protocol



- Commit:
  - Establish consensus throughout the views

View Change

- What if the **primary** is **faulty**??
  - non-faulty replicas detect the fault
  - replicas together start view change operation



View Changes

- View-change protocol provides **liveness**
  - Allow the system to make progress when primary fails
- If the primary fails, backups will not receive any message (such as PRE_PREPARE or COMMIT) from the primary
- View changes are triggered by timeouts
  - Prevent backups from waiting indefinitely for requests to execute

View Change

- Backup starts a timer when it receives a request, and the timer is not already running
  - The timer is stopped when the request is executed
  - Restarts when some new request comes

- If the timer expires at view v
  - Backup starts a view change to move the system to view $v + 1$

View Change

- On timer expiry, a backup stops accepting messages except
  - Checkpoint
  - View-change
  - New-View

View Change

- Multicasts a < VIEW_$\mathcal{C}$HANGE, v + 1, n, $\mathcal{C}$, $\mathcal{P}$, i >$_{,\sigma\_i}$ message to all replicas
  - n is the sequence number of the last stable checkpoint s known to i
  - $\mathcal{C}$ is a set of $2f$ + 1 valid checkpoint messages proving the correctness of s
  - $\mathcal{P}$ is a set containing a set $\mathcal{P}_m$ for each request m that prepared at i with a sequence number higher than n
    - Each set $\mathcal{P}_m$ contains a valid pre-prepare message and $2f$ matching

View Change

- The new view is initiated after receiving $2f$ view change messages

- The view change operation takes care of
  - Synchronization of checkpoints across the replicas
  - All the replicas are ready to start at the new view $v + 1$

Correctness

- **Safety:** The algorithm provides safety if all non-faulty replicas agree on the sequence numbers of requests that commit locally

Image Source: http://www.differencebetween.com/

Correctness

Liveness: To provide liveness, replicas must move to a new view if they are unable to execute a request

- A replica waits for $2f + 1$ view change messages and then starts a timer to initiate a new view (*avoid starting a view change too soon*)
- If a replica receives a set of $f + 1$ valid view change messages for views greater than its current view, it sends view change message (*prevents starting the next view change too late*)
- Faulty replicas are unable to impede progress by forcing frequent view change

For further details, look into

Castro, Miguel, and Barbara Liskov. "Practical Byzantine fault tolerance." *OSDI*. Vol. 99. 1999.

Consensus in Permissioned Model

- PBFT has well adopted in consensus for permissioned blockchain environments
  - Hyperledger
  - Tendermint Core

- Several scalability issues are still there, we'll discuss those in details in the later part of the our BCT course!


thank you!