

# CS 60002: Distributed Systems

## T9: Byzantine Agreement

Department of Computer Science  
and Engineering



INDIAN INSTITUTE OF TECHNOLOGY  
KHARAGPUR



Sandip Chakraborty  
[sandipc@cse.iitkgp.ac.in](mailto:sandipc@cse.iitkgp.ac.in)

# Byzantine Generals Problem



Commander



Lieutenant - 1

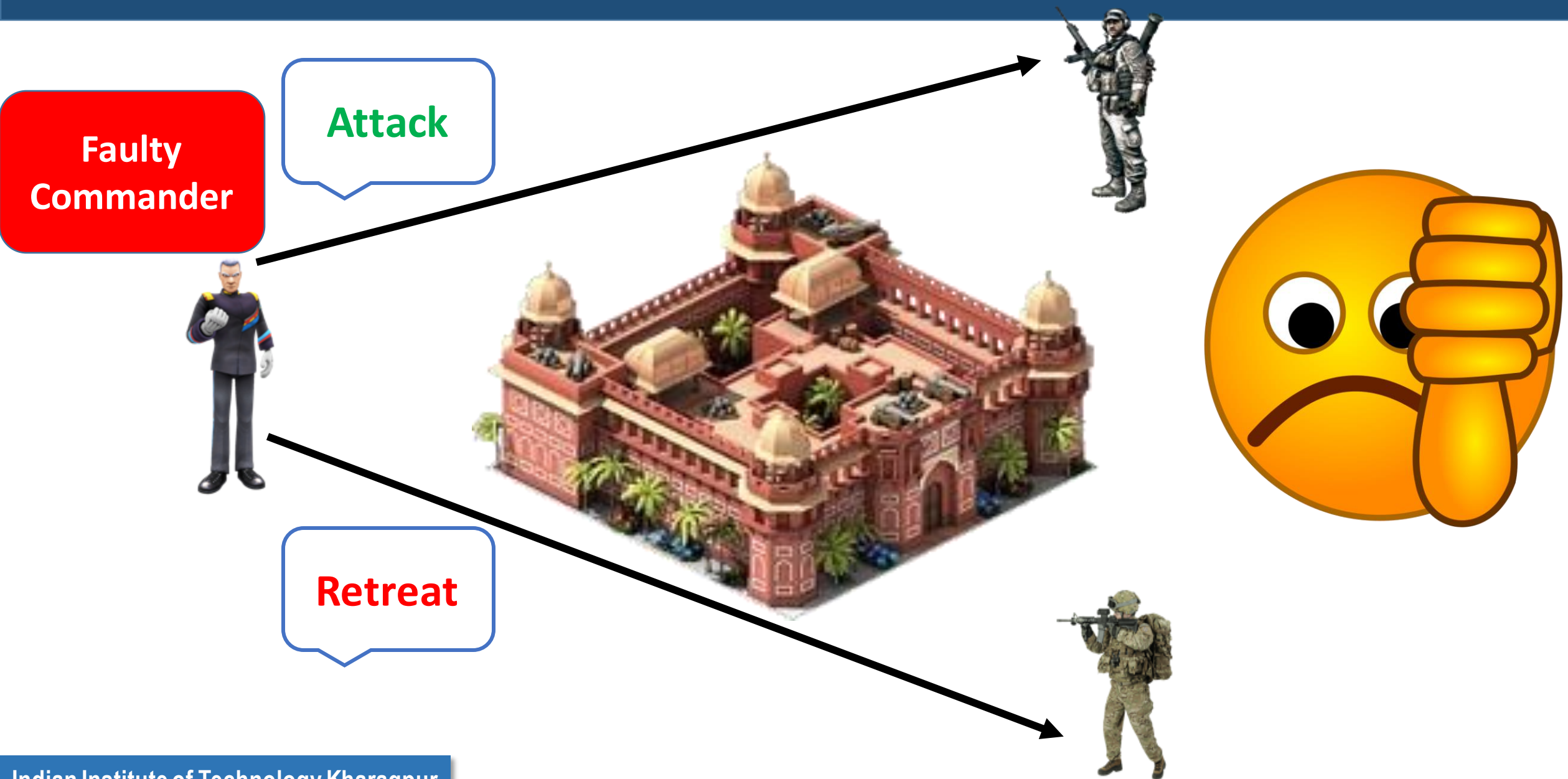


Lieutenant - 2

# Byzantine Generals Problem

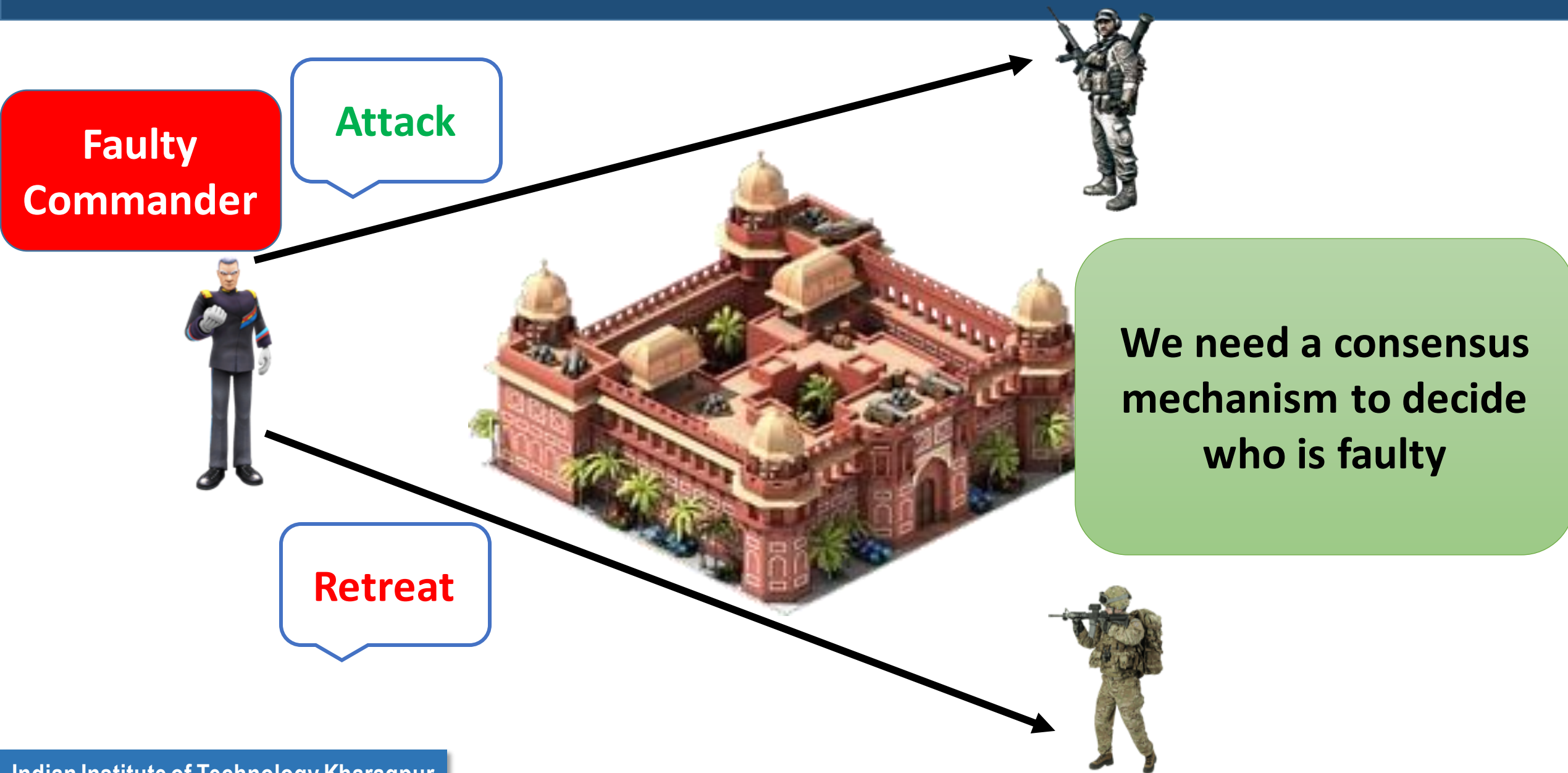


# Byzantine Generals Problem





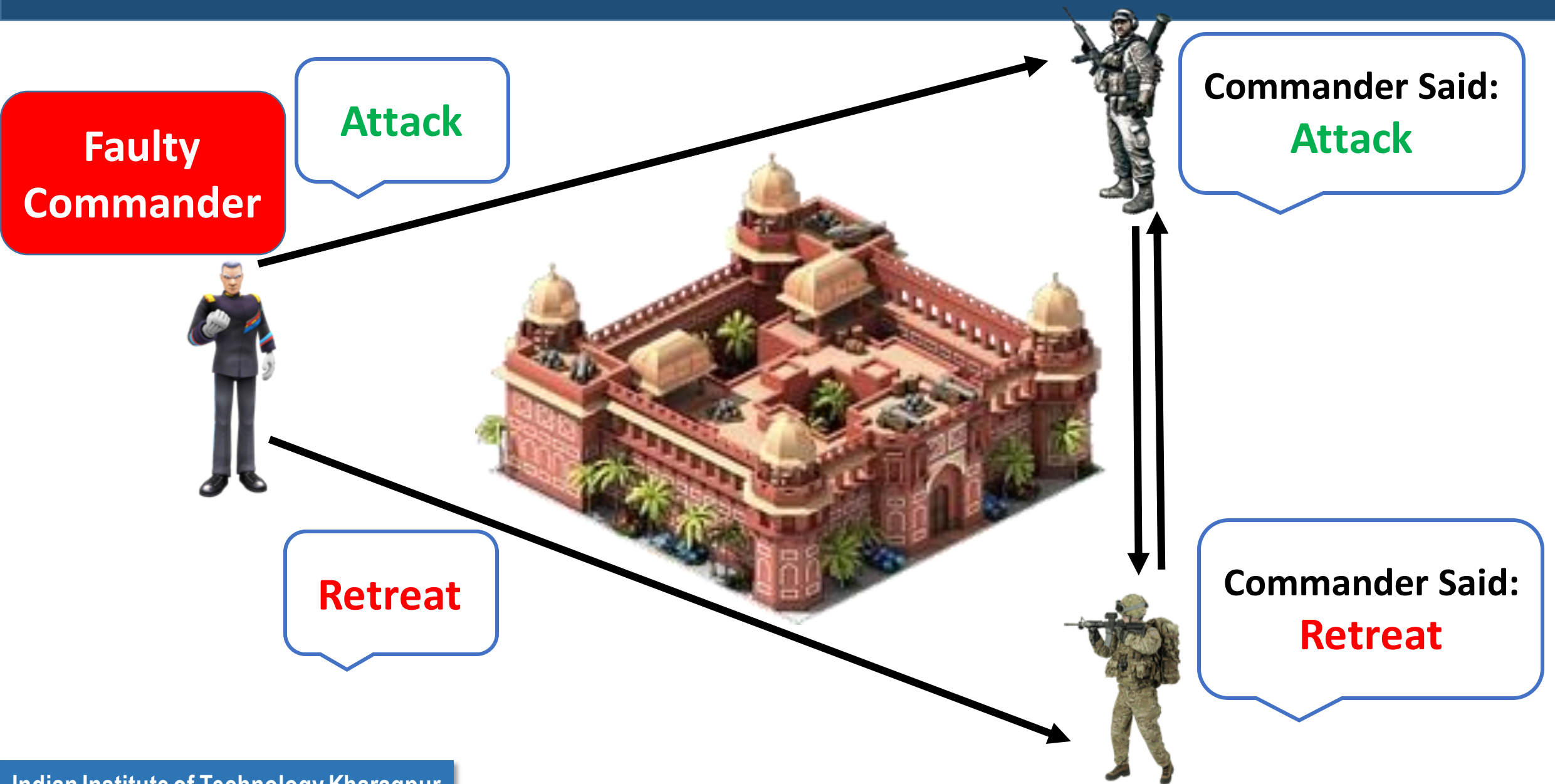
# Byzantine Generals Problem



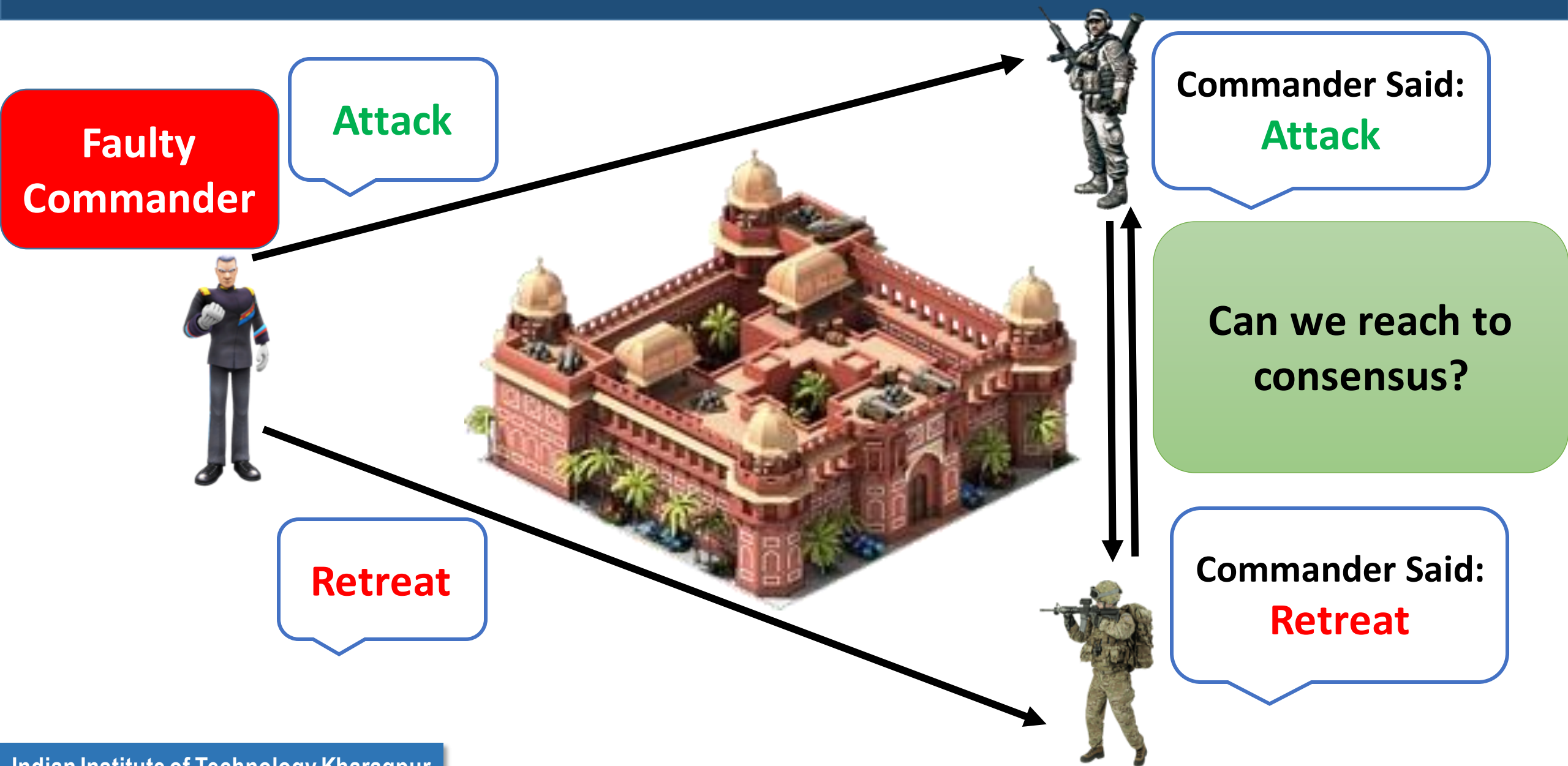
# Byzantine Generals Problem



# Byzantine Generals Problem



# Byzantine Generals Problem





# Byzantine Generals Problem

Good  
Commander

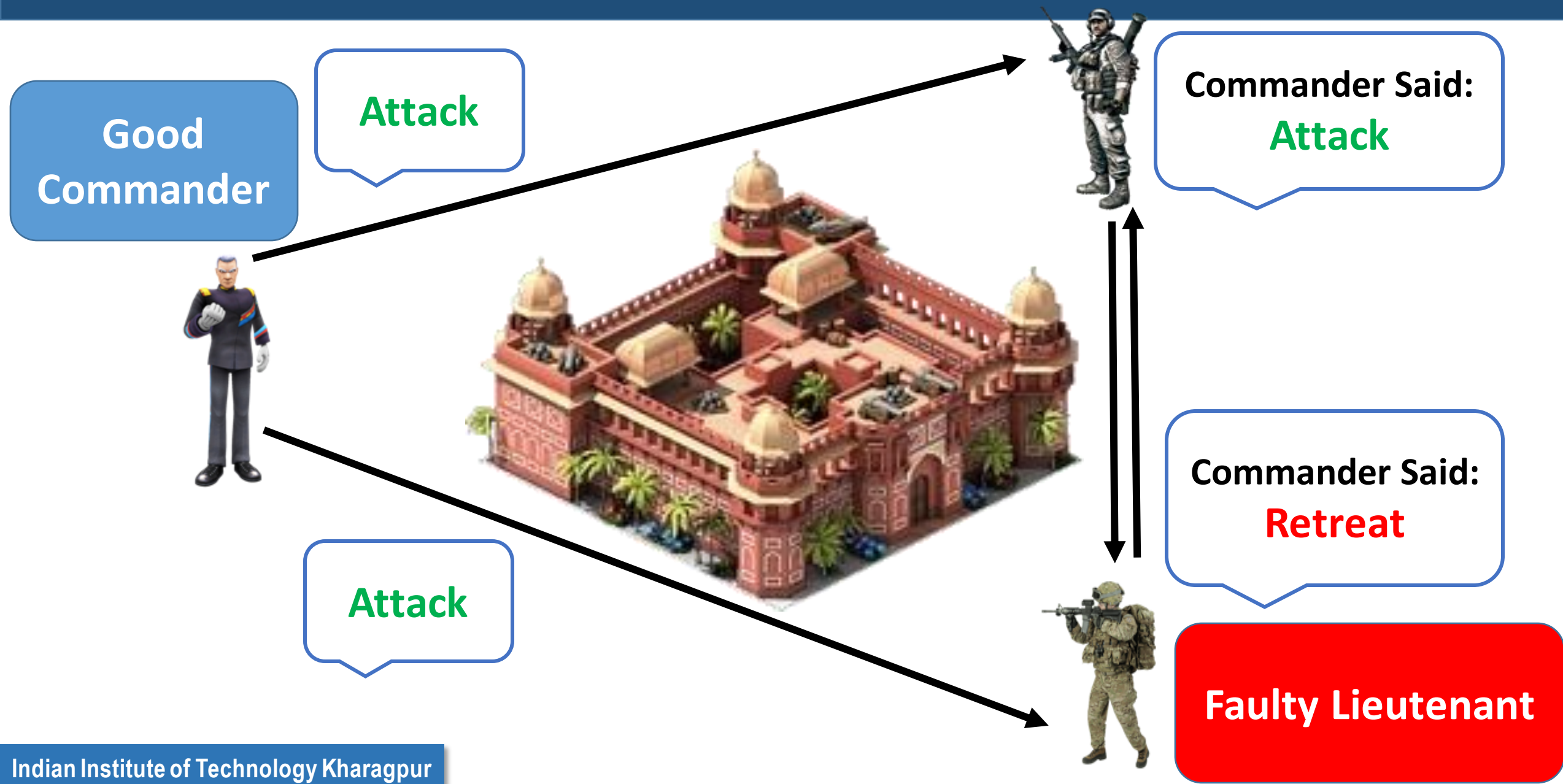
Attack



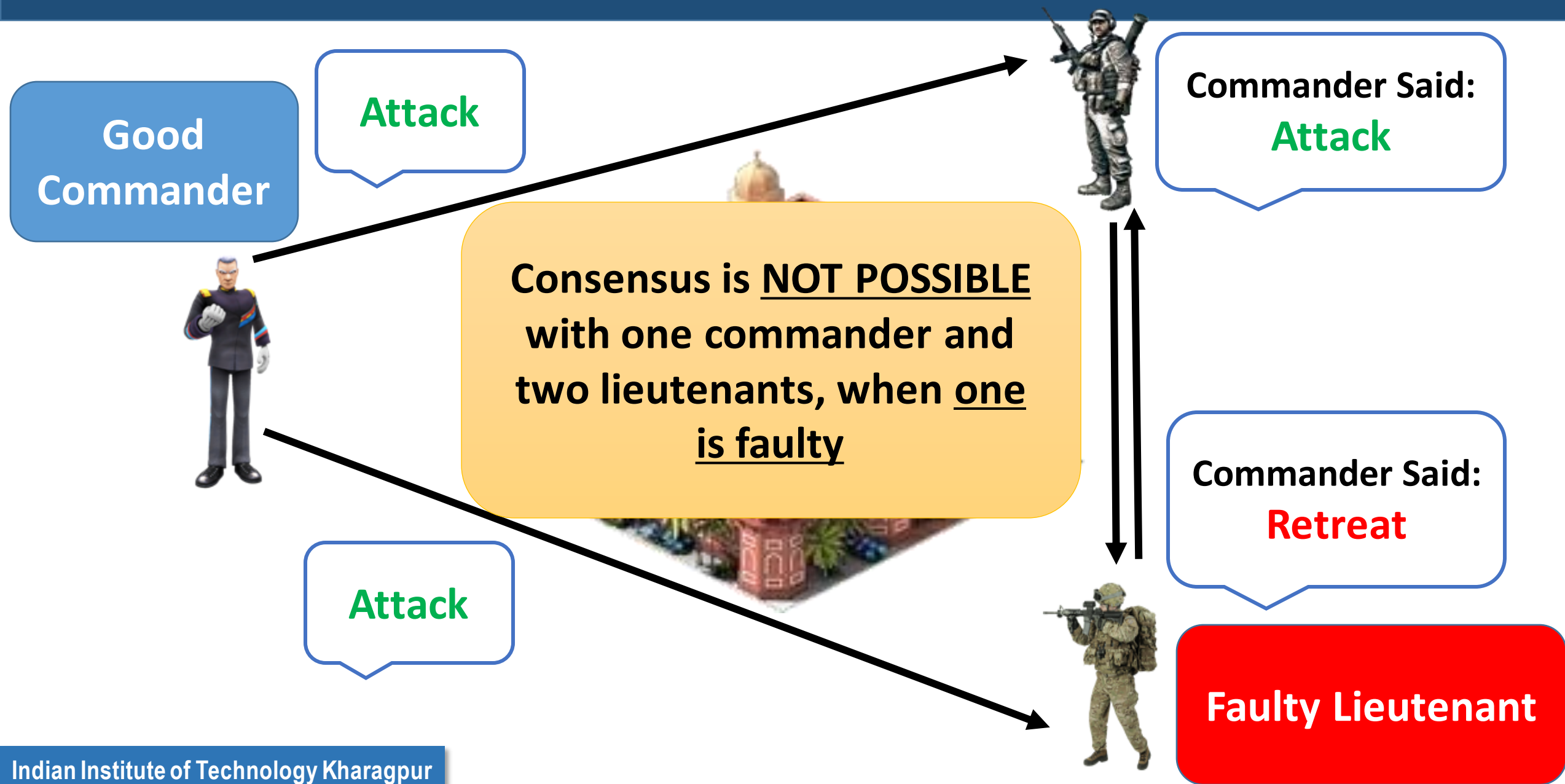
Attack



# Byzantine Generals Problem



# Byzantine Generals Problem



# Byzantine Generals Problem – Three Lieutenants





# Byzantine Generals Problem – Three Lieutenants

**Faulty  
Commander**

**Attack**

**Attack**

**Retreat**



# Byzantine Generals Problem – Three Lieutenants

**Faulty  
Commander**

**Attack**

**Attack**

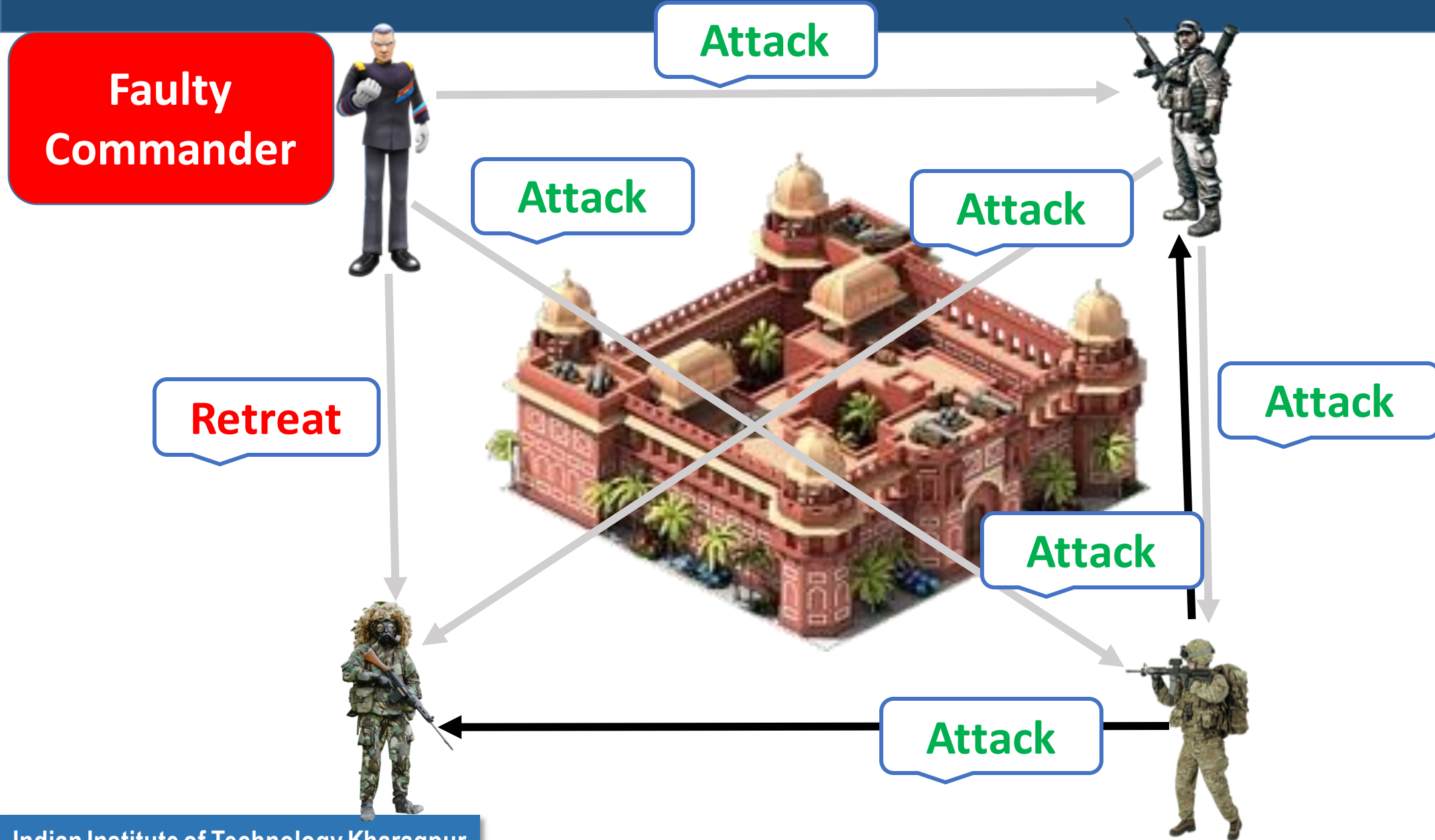
**Attack**

**Retreat**

**Attack**



# Byzantine Generals Problem – Three Lieutenants



**Faulty  
Commander**





# Byzantine Generals Problem – Three Lieutenants

**Faulty  
Commander**

**Attack**

**Attack: 3  
Retreat: 0**

**Attack**

**Attack**

**Retreat**

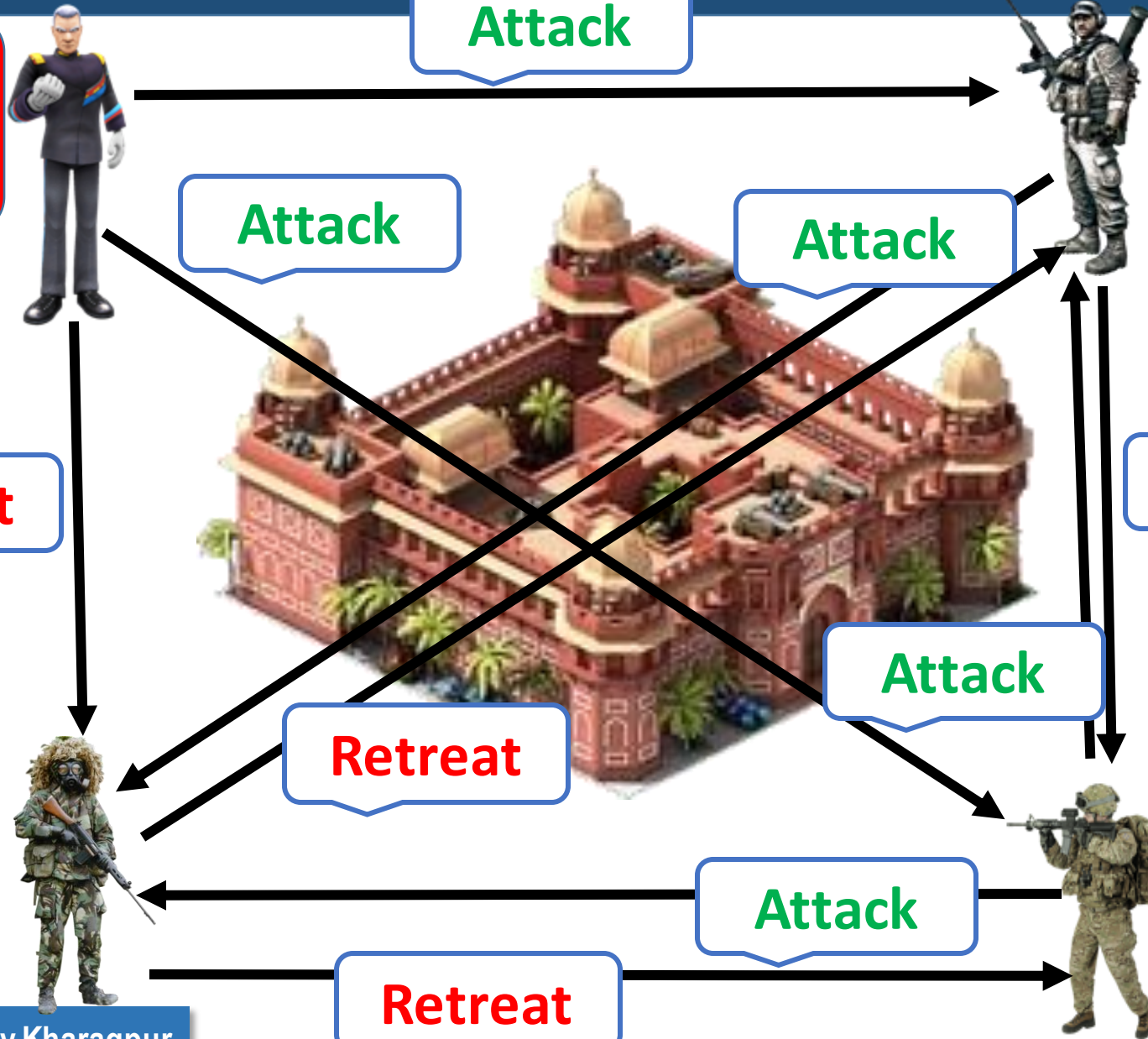
**Attack**

**Attack**

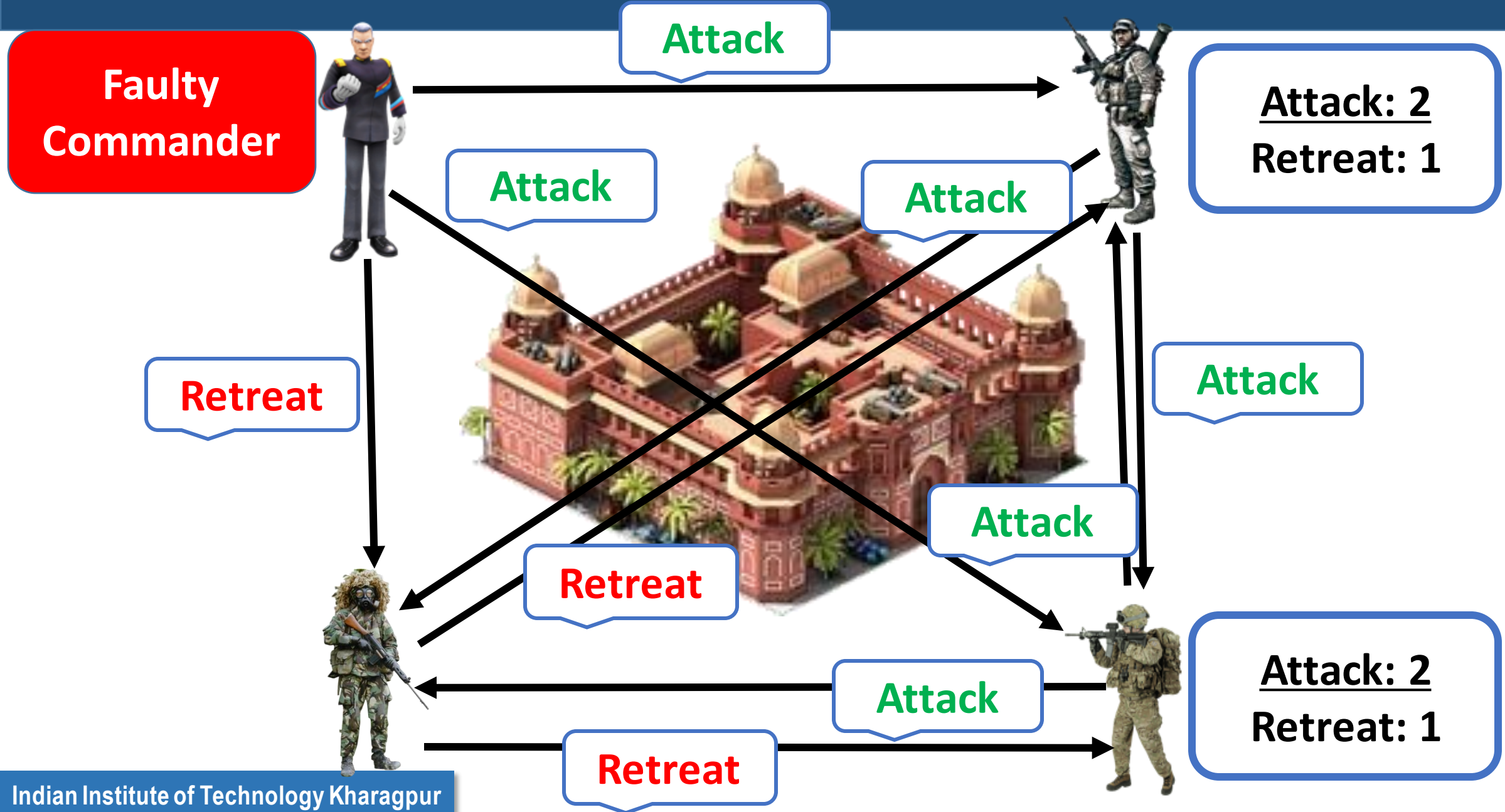
**Retreat**

**Attack**

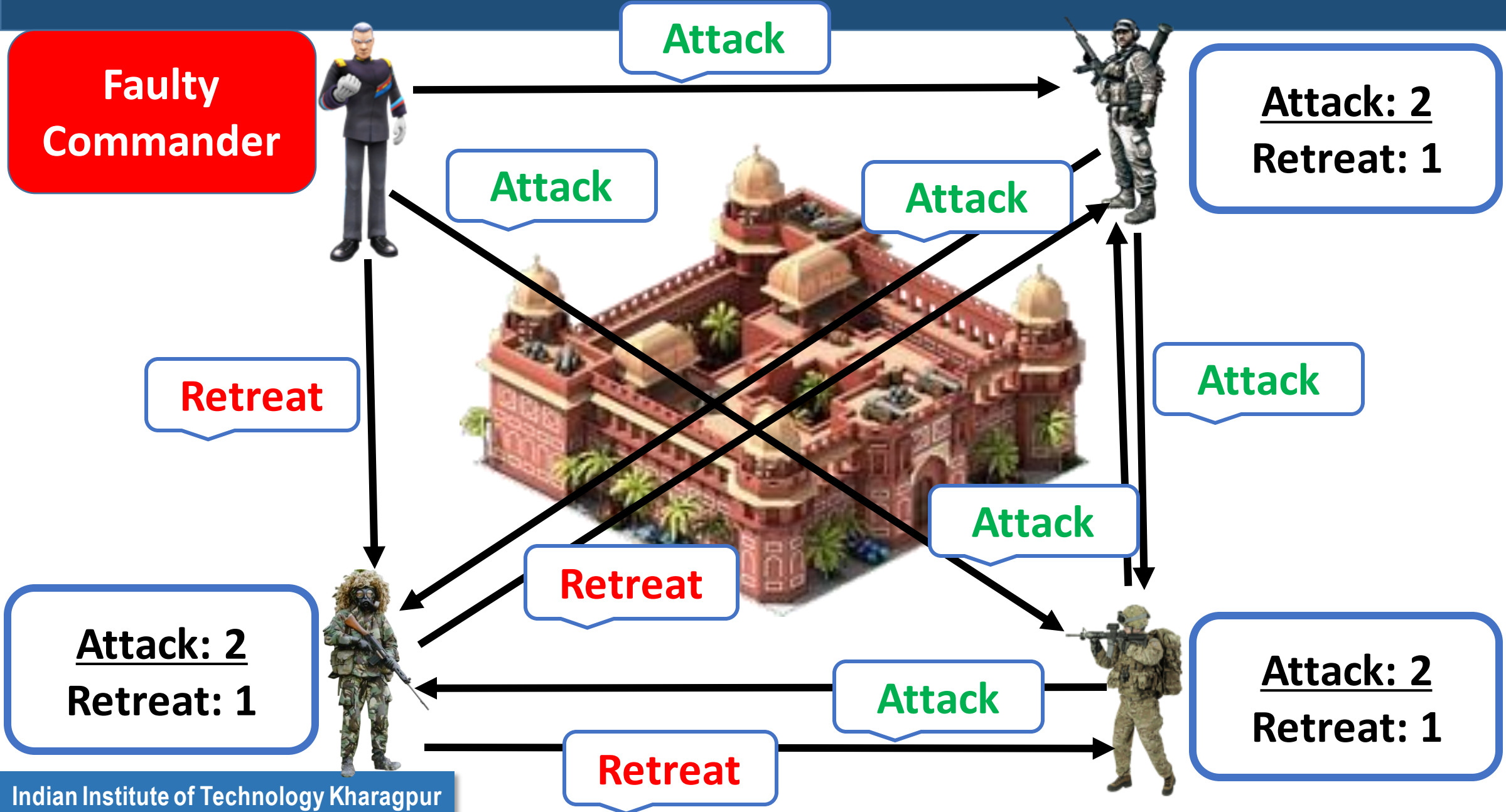
**Retreat**



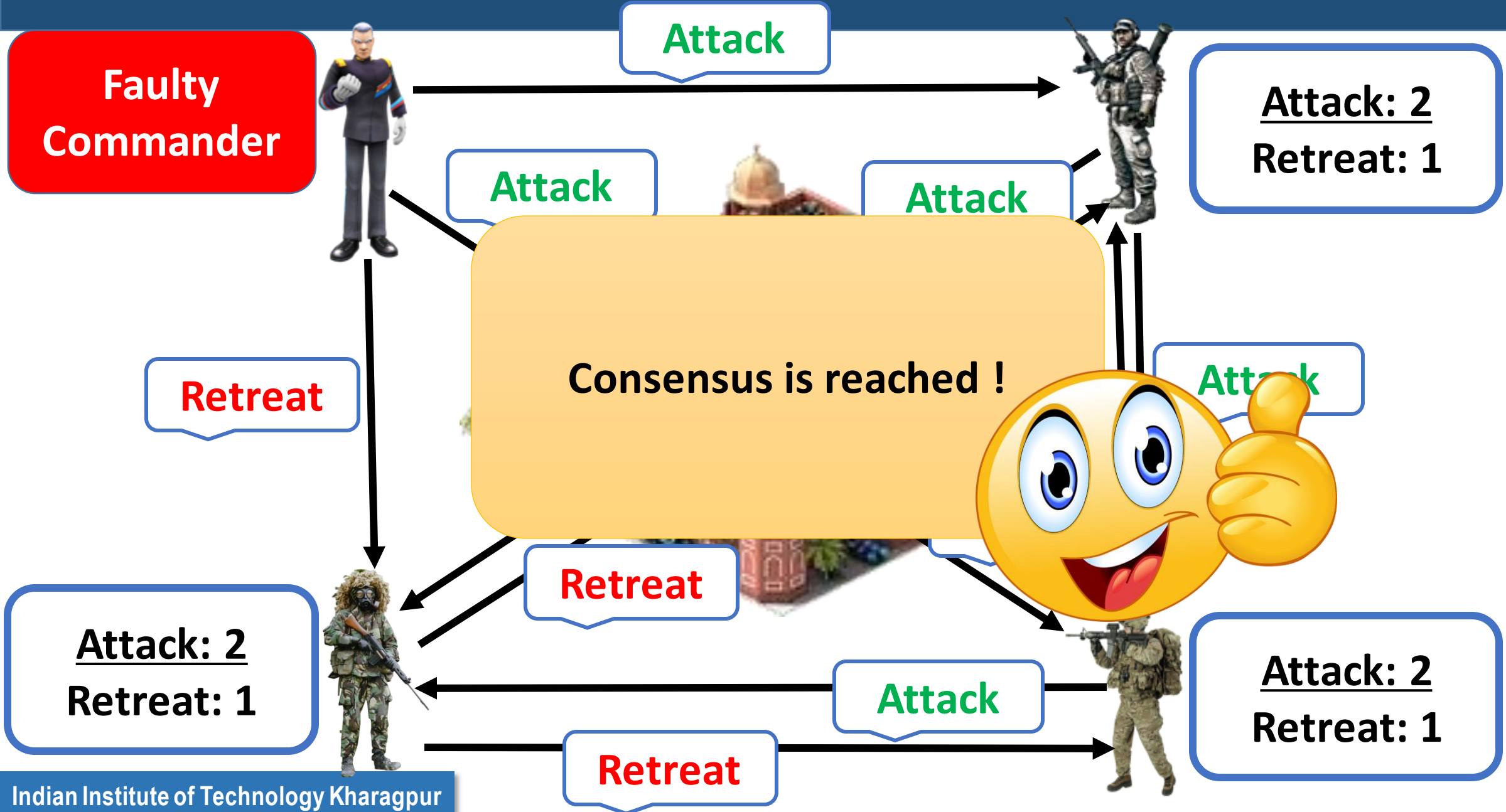
# Byzantine Generals Problem – Three Lieutenants



# Byzantine Generals Problem – Three Lieutenants



# Byzantine Generals Problem – Three Lieutenants





# Byzantine Generals Problem – Three Lieutenants

Good  
Commander

Attack

Attack

Attack



# Byzantine Generals Problem – Three Lieutenants

Good  
Commander

Attack

Attack

Attack

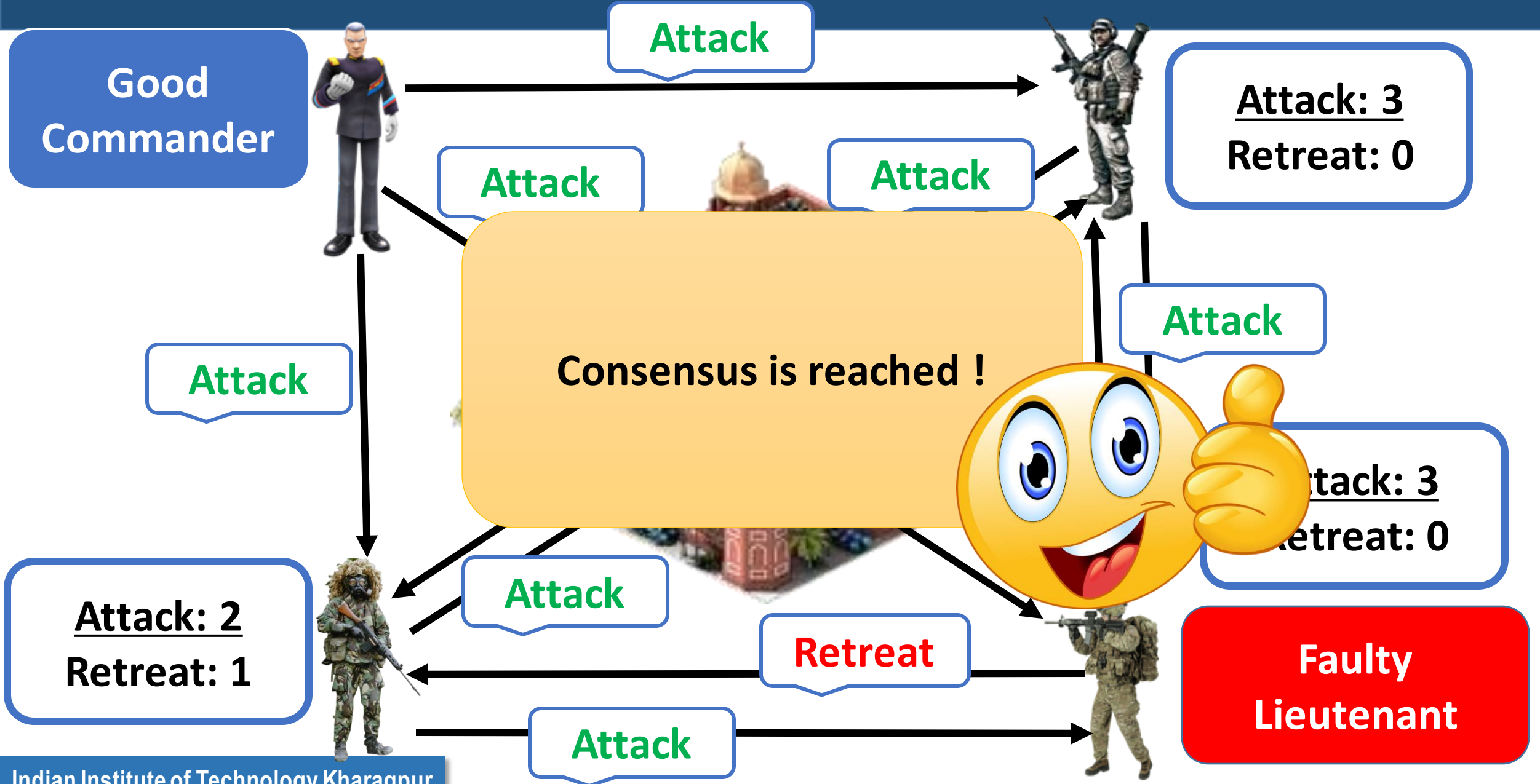
Attack

Retreat

Faulty  
Lieutenant



# Byzantine Generals Problem – Three Lieutenants



# A Formal Definition

- A commander must send an order to his  $n-1$  lieutenants so that the following two conditions are met
  - **IC1:** All loyal lieutenants obey the same order
  - **IC2:** If the commander is loyal, then every loyal lieutenants obey the order that he sends
- IC1 and IC2 are called *interactive consistency* conditions

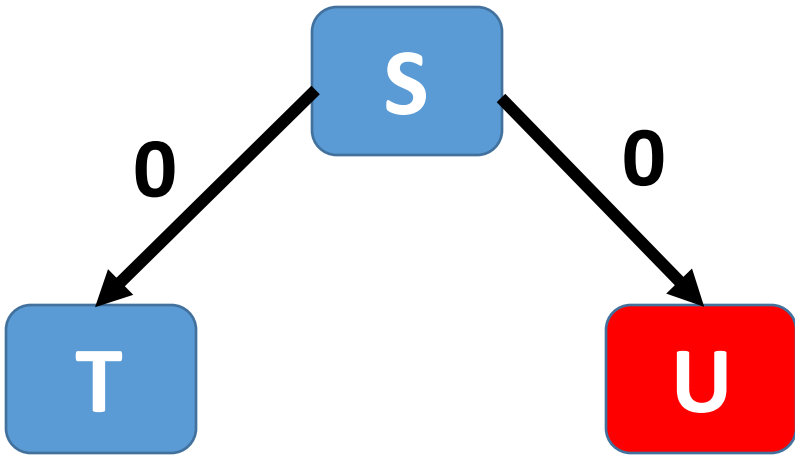


# Impossibility Results

- Let  $F$  be the maximum number of faulty processes that the protocol can tolerate – **Byzantine agreement is not possible with fewer than  $3F+1$  processes**

# Impossibility Results

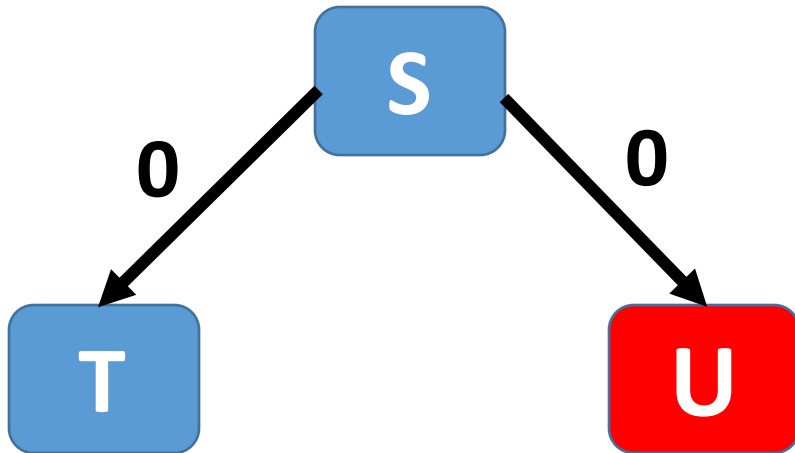
- Let  $F$  be the maximum number of faulty processes that the protocol can tolerate – **Byzantine agreement is not possible with fewer than  $3F+1$  processes**



**Case 0:**  $S$  is a correct process that sends 0

# Impossibility Results

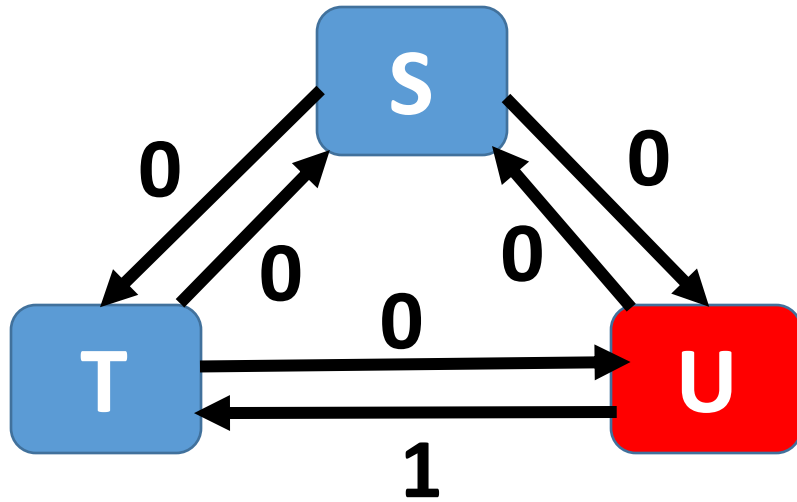
- Let  $F$  be the maximum number of faulty processes that the protocol can tolerate – **Byzantine agreement is not possible with fewer than  $3F+1$  processes**



**Case 0:** **S** is a correct process that sends 0  
**T** is correct, **T** must decide 0 (**validity**)

# Impossibility Results

- Let  $F$  be the maximum number of faulty processes that the protocol can tolerate – **Byzantine agreement is not possible with fewer than  $3F+1$  processes**

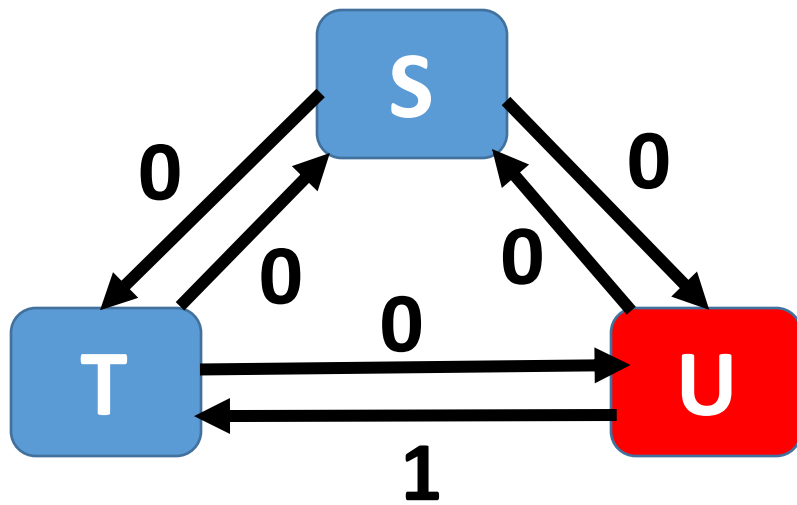


**Case 0:** S is a correct process that sends 0  
T is correct, T must decide 0 (**validity**)

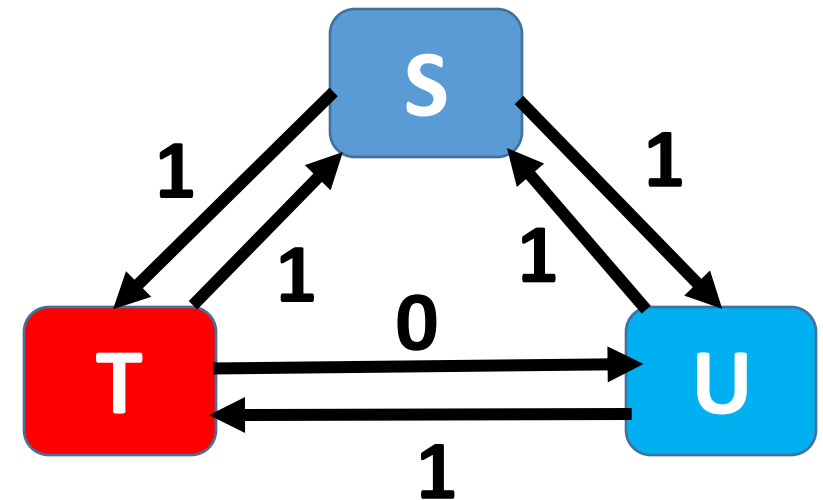


# Impossibility Results

- Let  $F$  be the maximum number of faulty processes that the protocol can tolerate – **Byzantine agreement is not possible with fewer than  $3F+1$  processes**



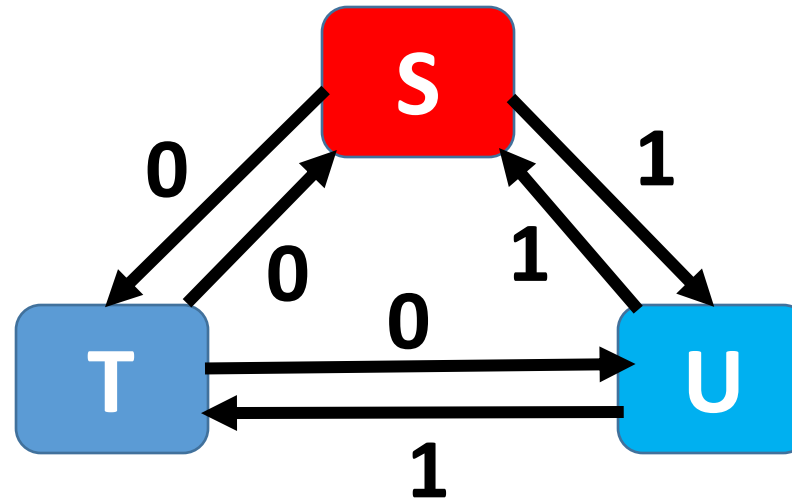
**Case 0:** S is a correct process that sends 0  
T is correct, T must decide 0 (**validity**)



**Case 1:** S is a correct process that sends 1  
U is correct, U must decide 1 (**validity**)

# Impossibility Results

- Let  $F$  be the maximum number of faulty processes that the protocol can tolerate – **Byzantine agreement is not possible with fewer than  $3F+1$  processes**



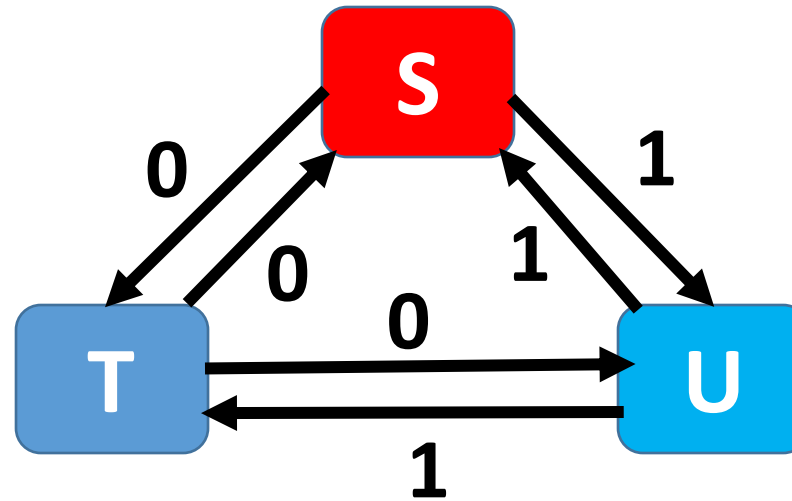
**Case 2: S is faulty**

**For T: Same as Case-0**

**For U: Same as Case-1**

# Impossibility Results

- Let  $F$  be the maximum number of faulty processes that the protocol can tolerate – **Byzantine agreement is not possible with fewer than  $3F+1$  processes**



A correct process cannot distinguish between the two cases!

Case 2: S is faulty

For T: Same as Case-0

For U: Same as Case-1

# Impossibility Results

- Let  $F$  be the maximum number of faulty processes that the protocol can tolerate – **Byzantine agreement is not possible with fewer than  $3F+1$  processes**
- **NOTE:** Byzantine faults can be detected with fewer than  $3F+1$  processes
  - Needs  $F+1$  processes to detect Byzantine faults – a correct node can reliably detect faults irrespective of the fraction of faulty nodes.
  - Check the following for a formal proof and the design of a Byzantine fault detector -- [https://www.usenix.org/legacy/event/hotdep06/tech/prelim\\_papers/haeberlen/haeberlen\\_html/index.html](https://www.usenix.org/legacy/event/hotdep06/tech/prelim_papers/haeberlen/haeberlen_html/index.html)



# Impossibility Results

- Let  $F$  be the maximum number of faulty processes that the protocol can tolerate – **Byzantine agreement is not possible with fewer than  $3F+1$  processes**
- **NOTE:** Byzantine faults can be detected with fewer than  $3F+1$  processes
  - Needs  $F+1$  processes to detect Byzantine faults – a correct node can reliably detect faults irrespective of the fraction of faulty nodes.
  - Check the following for a formal proof and the design of a Byzantine fault detector -- [https://www.usenix.org/legacy/event/hotdep06/tech/prelim\\_papers/haeberlen/haeberlen\\_html/index.html](https://www.usenix.org/legacy/event/hotdep06/tech/prelim_papers/haeberlen/haeberlen_html/index.html)
  - **This is indeed interesting, many works have taken an alternate approach of BFT consensus – detect the faulty nodes and then throw them out – less number of rounds but implementation is sometime complex.**

# Asynchronous Byzantine Agreement

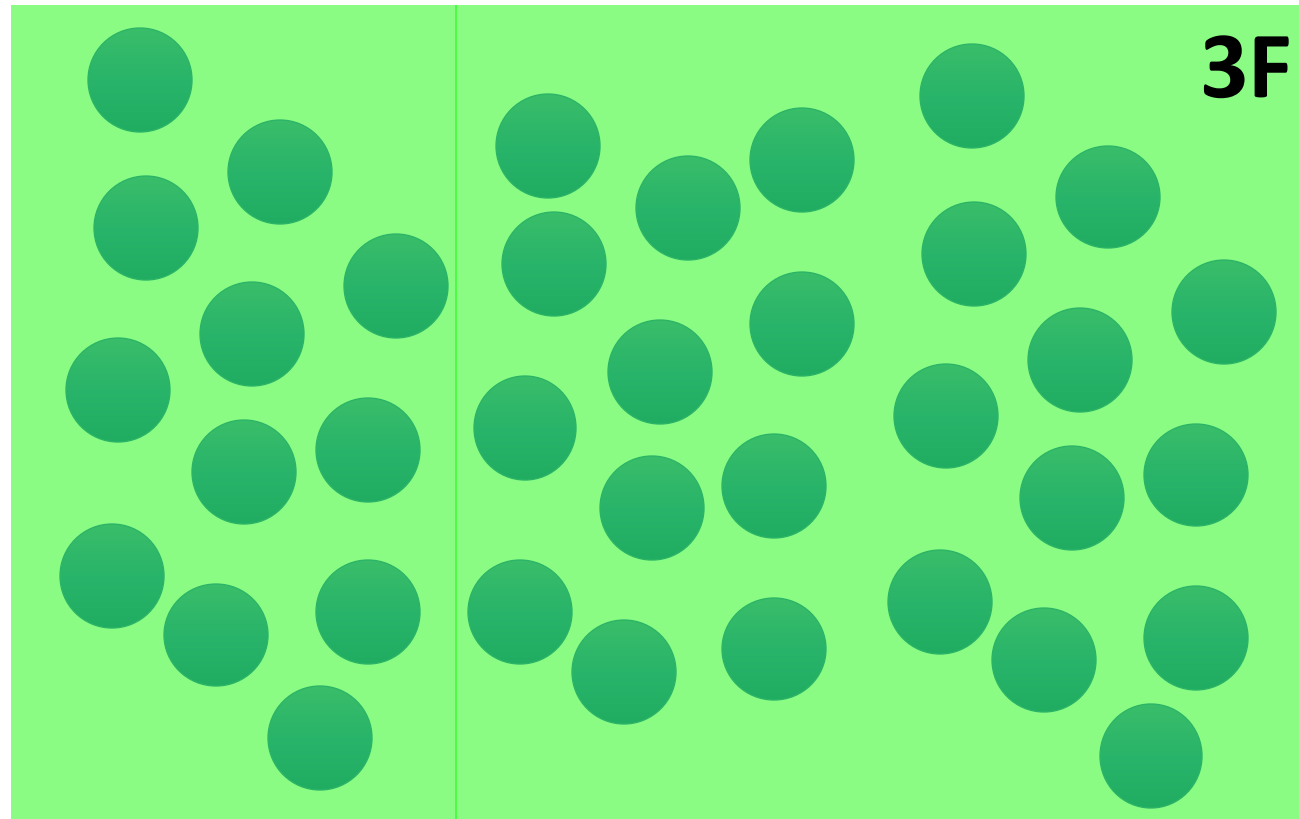
- $F$  faulty nodes – need  $3F + 1$  nodes to reach consensus

# Asynchronous Byzantine Agreement

- $F$  faulty nodes – need  $3F + 1$  nodes to reach consensus
  - Faulty nodes create partition in the network

# Asynchronous Byzantine Agreement

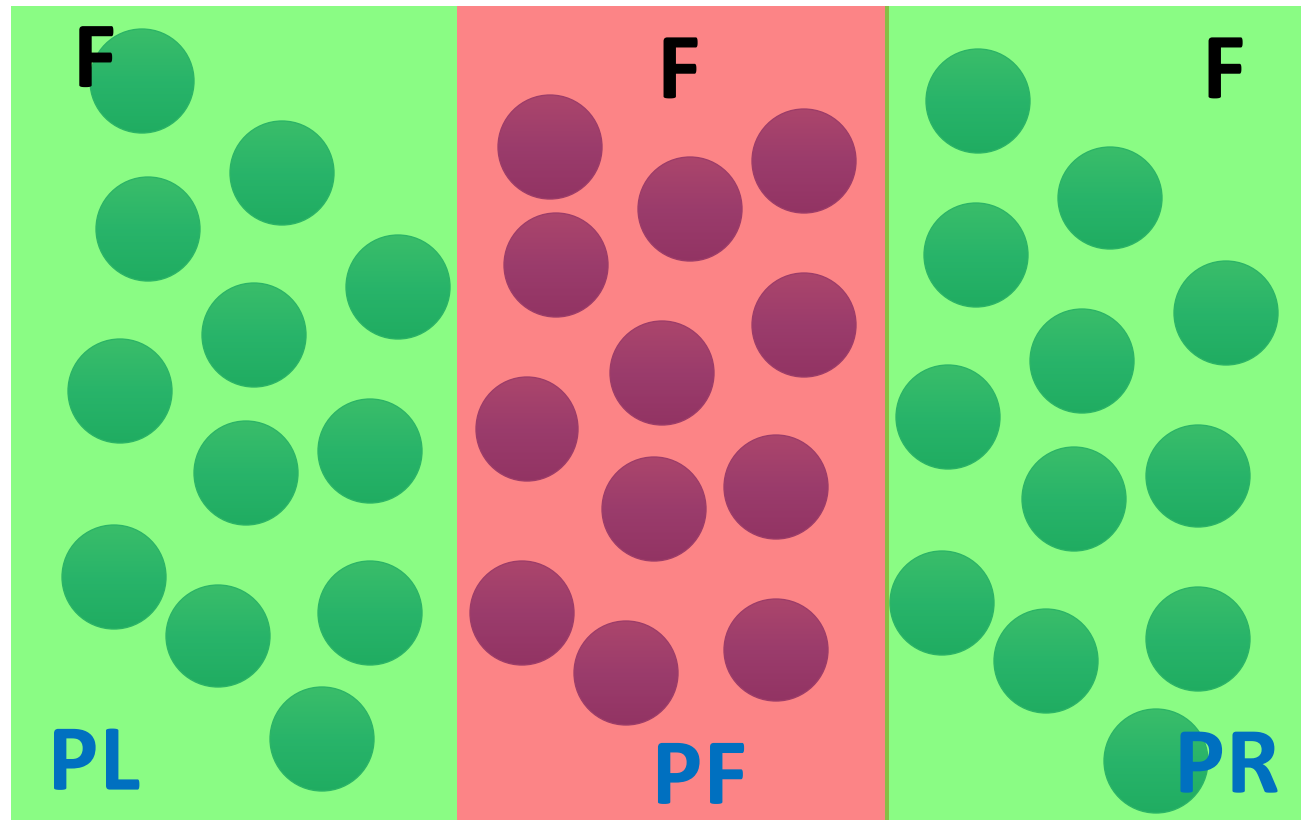
- $F$  faulty nodes – need  $3F + 1$  nodes to reach consensus
  - Faulty nodes create partition in the network





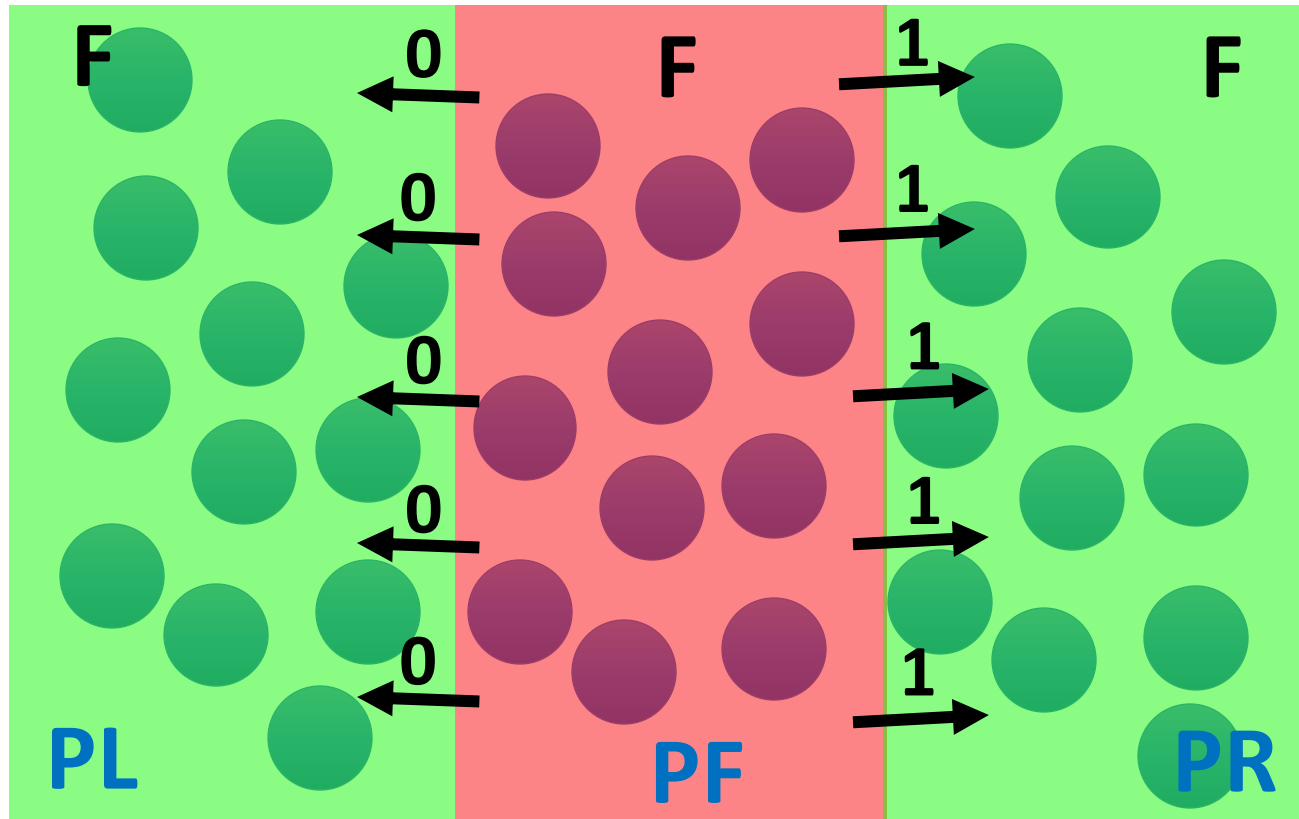
# Asynchronous Byzantine Agreement

- $F$  faulty nodes – need  $3F + 1$  nodes to reach consensus
  - Faulty nodes create partition in the network



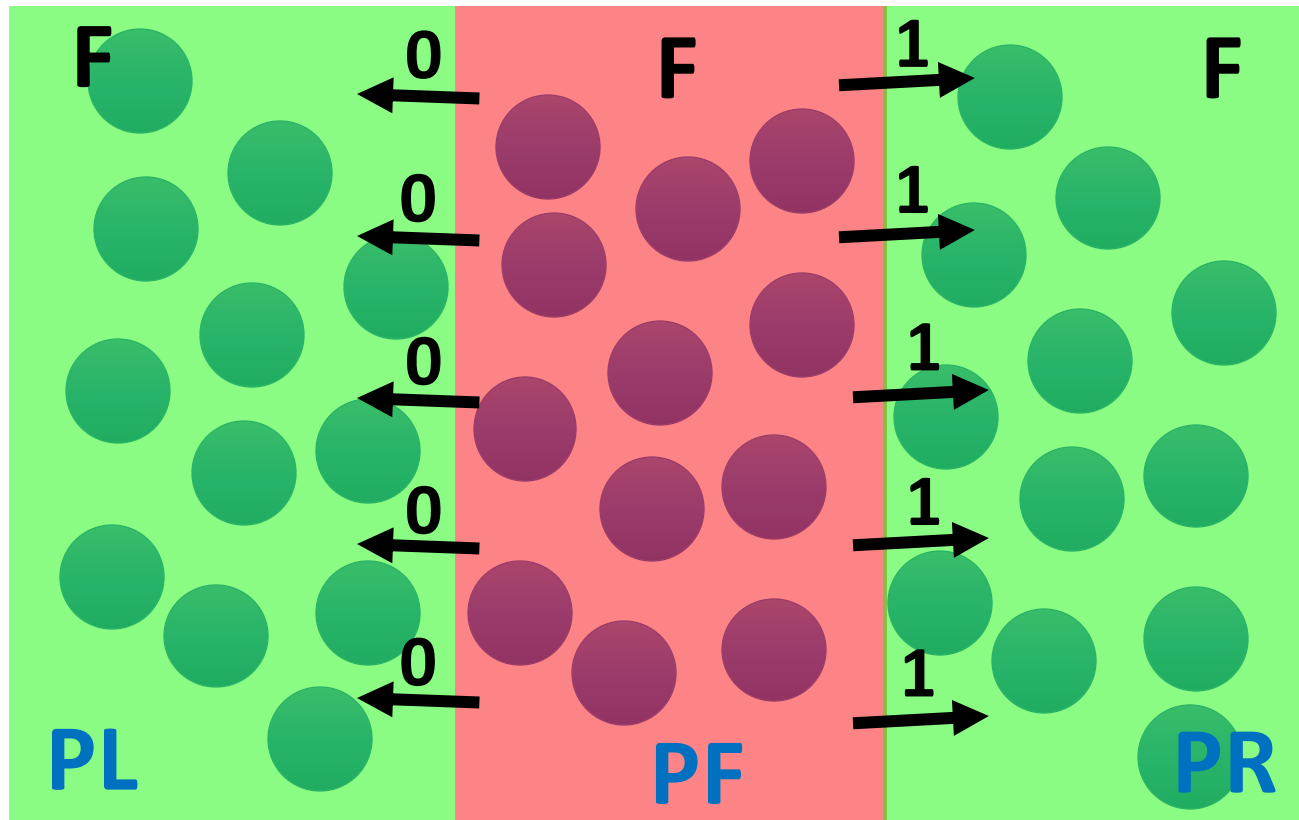
# Asynchronous Byzantine Agreement

- $F$  faulty nodes – need  $3F + 1$  nodes to reach consensus
  - Faulty nodes create partition in the network



# Asynchronous Byzantine Agreement

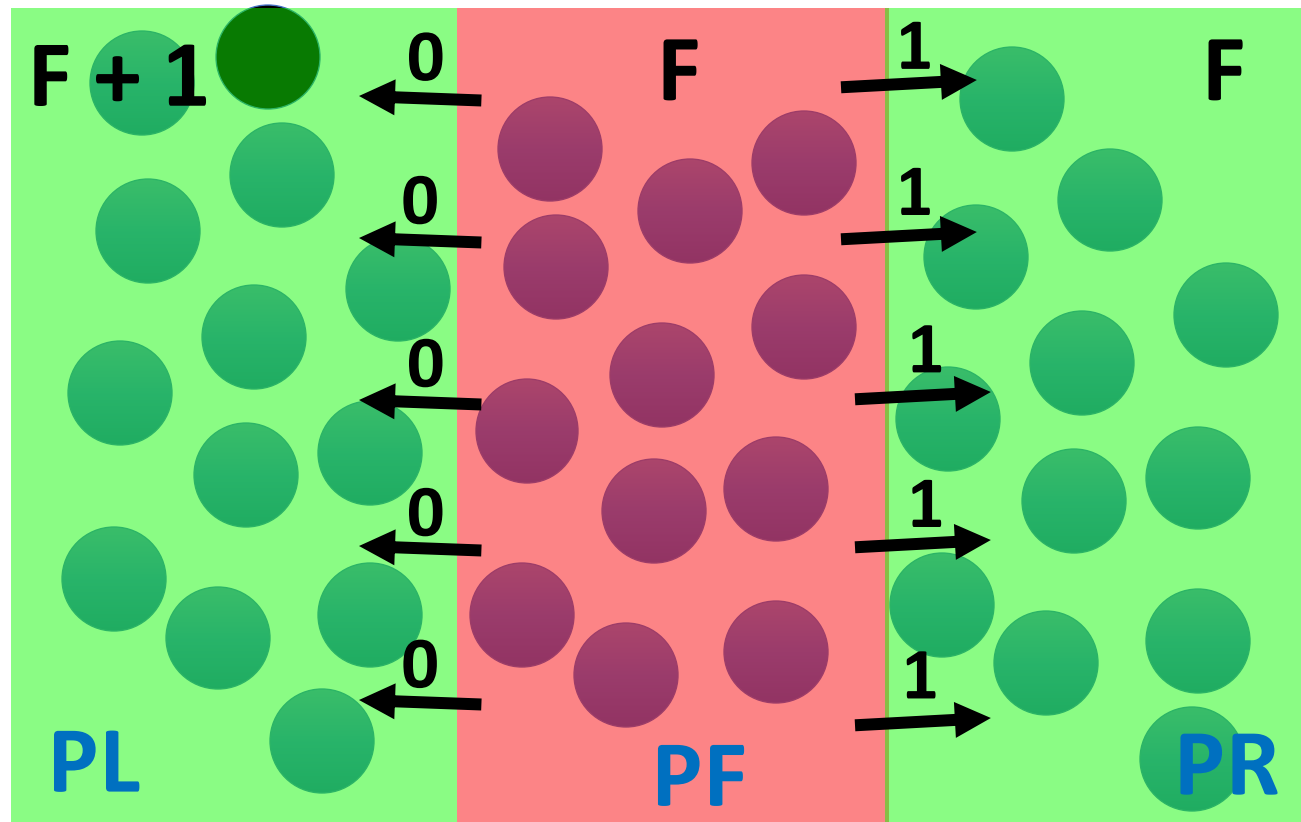
- $F$  faulty nodes – need  $3F + 1$  nodes to reach consensus
  - Faulty nodes create partition in the network



**Either PL or PR  
must break the  
tie**

# Asynchronous Byzantine Agreement

- $F$  faulty nodes – need  $3F + 1$  nodes to reach consensus
  - Faulty nodes create partition in the network



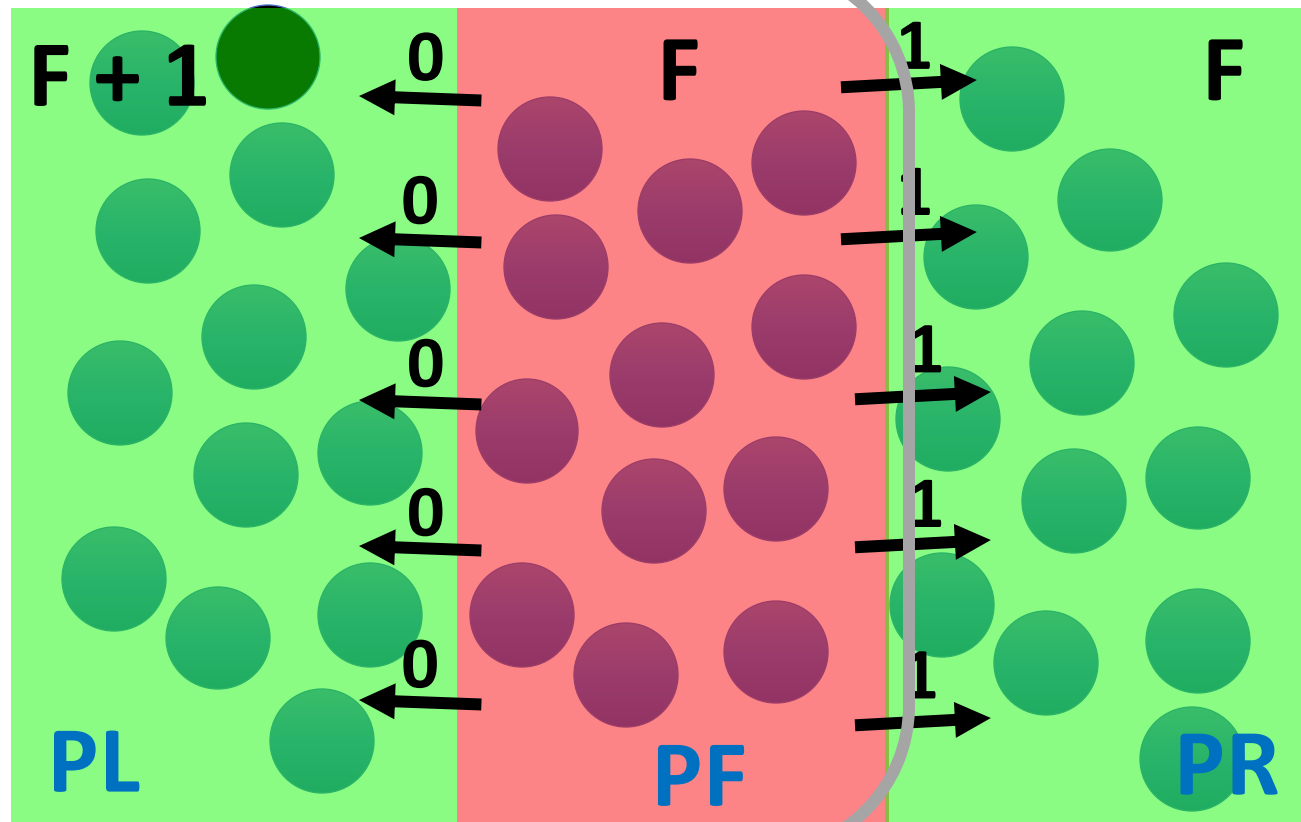
Put one  
additional node  
to PL / PR



# Asynchronous Byzantine Agreement

- $F$  faulty nodes – need  $3F + 1$  nodes to reach consensus
  - Faulty nodes create partition in the network

Breaks the  
tie to reach  
consensus



Put one  
additional node  
to PL / PR

# Broad Idea of Agreement Protocols

- **Assumptions:** Synchronous, Completely connected, reliable communication, crash faults (crash-stop), maximum  $f$  faults
- $(f+1)$  rounds for each process
- **Round 1:**
  - Broadcast own value to other processes
- **Round 2 to Round  $f+1$ :**
  - Broadcast any new received values
- **At the end of Round  $f+1$ :**
  - Decide on the minimum value received

# Broad Idea of Agreement Protocols

- With  $f+1$  rounds, there is at least one round with no faults
  - At the end of this round, all non-faulty processes have the values of all other non-faulty processes

# Broad Idea of Agreement Protocols

- With  $f+1$  rounds, there is at least one round with no faults
  - At the end of this round, all non-faulty processes have the values of all other non-faulty processes
- For a faulty process
  - If it failed before sending its value to any other node, no one has its value
  - If it sent to some nodes before failing, at the end of the round with no failures, all non-faulty nodes will have its value

# Broad Idea of Agreement Protocols

- With  $f+1$  rounds, there is at least one round with no faults
  - At the end of this round, all non-faulty processes have the values of all other non-faulty processes
- For a faulty process
  - If it failed before sending its value to any other node, no one has its value
  - If it sent to some nodes before failing, at the end of the round with no failures, all non-faulty nodes will have its value
- However, we do not know a priori when that round will come, so we have to go for  $f+1$  rounds



# Broad Idea of Agreement Protocols

- With  $f+1$  rounds, there is at least one round with no faults
  - At the end of this round, all non-faulty processes have the values of all other non-faulty processes
- For a faulty process
  - If it failed before sending its value to any other node, no one has its value
  - If it sent to some nodes before failing, at the end of the round with no failures, all non-faulty nodes will have its value
- However, we do not know a priori when that round will come, so we have to go for  $f+1$  rounds – **this minimum bound is also applicable for Byzantine faults, try the proof!**

# BFT Consensus

- **Lamport-Shostak-Peas Algorithm\***

- Synchronous environment
- Reliable communication channel
- Fully Connected Network
- Receivers always know the identity of the Senders

\* LAMPORT, LESLIE, ROBERT SHOSTAK, and MARSHALL PEASE. "The Byzantine Generals Problem." *ACM Transactions on Programming Languages and Systems* 4.3 (1982): 382-401.

# BFT Consensus

- **Lamport-Shostak-Peas Algorithm\***

- Synchronous environment
- Reliable communication channel
- Fully Connected Network
- Receivers always know the identity of the Senders

**Unrealistic assumptions for  
real networks**

\* LAMPORT, LESLIE, ROBERT SHOSTAK, and MARSHALL PEASE. "The Byzantine Generals Problem." *ACM Transactions on Programming Languages and Systems* 4.3 (1982): 382-401.

# BFT Consensus

- **Lamport-Shostak-Peas Algorithm\***
  - Synchronous environment
  - Reliable communication channel
  - Fully Connected Network
  - Receivers always know the identity of the Senders
- Many different variants of BFT Consensus have emerged

# BFT Consensus

- **Lamport-Shostak-Peas Algorithm\***
  - Synchronous environment
  - Reliable communication channel
  - Fully Connected Network
  - Receivers always know the identity of the Senders
- Many different variants of BFT Consensus have emerged
- **Practical Byzantine Fault Tolerance (PBFT)\*\***
  - Use cryptographic techniques to release the *\*unrealistic\** assumptions

\*\* Castro, Miguel, and Barbara Liskov. "Practical byzantine fault tolerance." *USENIX OSDI*. Vol. 99. No. 1999. 1999.

# Lamport-Shostak-Peas Algorithm

- **Oral messages** – messages can be forged or changed in any manner, but the receiver always knows the sender
- The algorithm has been defined in a recursive way



# Lamport-Shostak-Peas Algorithm

## OM(f) for $f > 0$

- Source  $x$  broadcast values to all processes
- Let  $v_i$  = the value received by process  $i$  from source (0 if no value received). Process  $i$  acts as a new source and initiates OM( $f-1$ ), sending  $v_i$  to remaining  $f-2$  processes
- For each  $i, j, i \neq j$ , let  $v_{ij}$  = value received by process  $i$  from process  $j$  in the previous step using OM( $f-1$ ). Process  $i$  uses the value majority ( $v_{i1}, v_{i2}, \dots, v_{in}$ )

# Lamport-Shostak-Peas Algorithm

## OM(0)

- Source  $x$  broadcast values to all processes
- Each process uses the value; if no value received, 0 is used.

# Lamport-Shostak-Peas Algorithm

## OM(0)

- Source  $x$  broadcast values to all processes
- Each process uses the value; if no value received, 0 is used.
- Time complexity:  $(f+1)$  rounds
- Message complexity:  $O(n^f)$  where  $n$  is the total number of processes

# Practical Byzantine Fault Tolerance

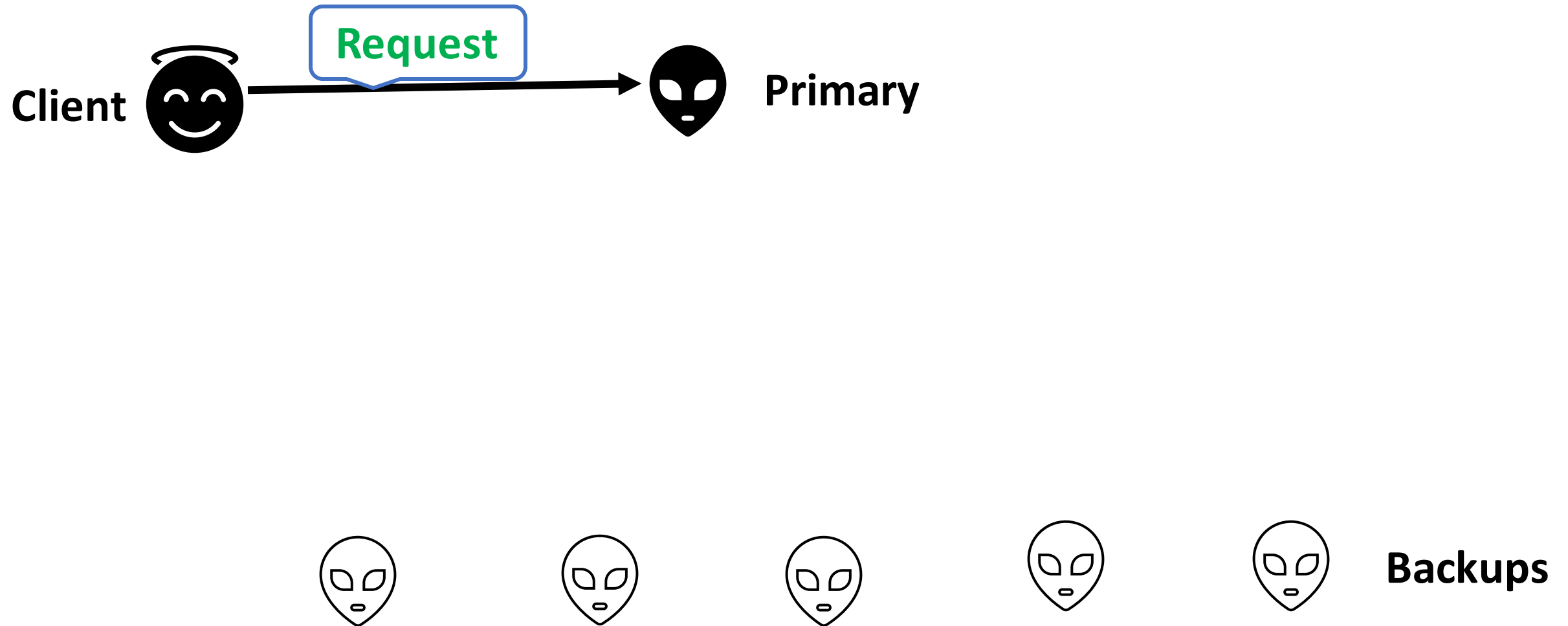
- **Why Practical?**

- Considers an asynchronous environment (Gives priority to Safety over Liveness)
  - Utilizes digital signature to validate the identity of the senders
  - Low overhead
- Incorporated in many distributed applications including Blockchain
- Uses cryptographic techniques to make the messages tamper-proof

# PBFT Overview

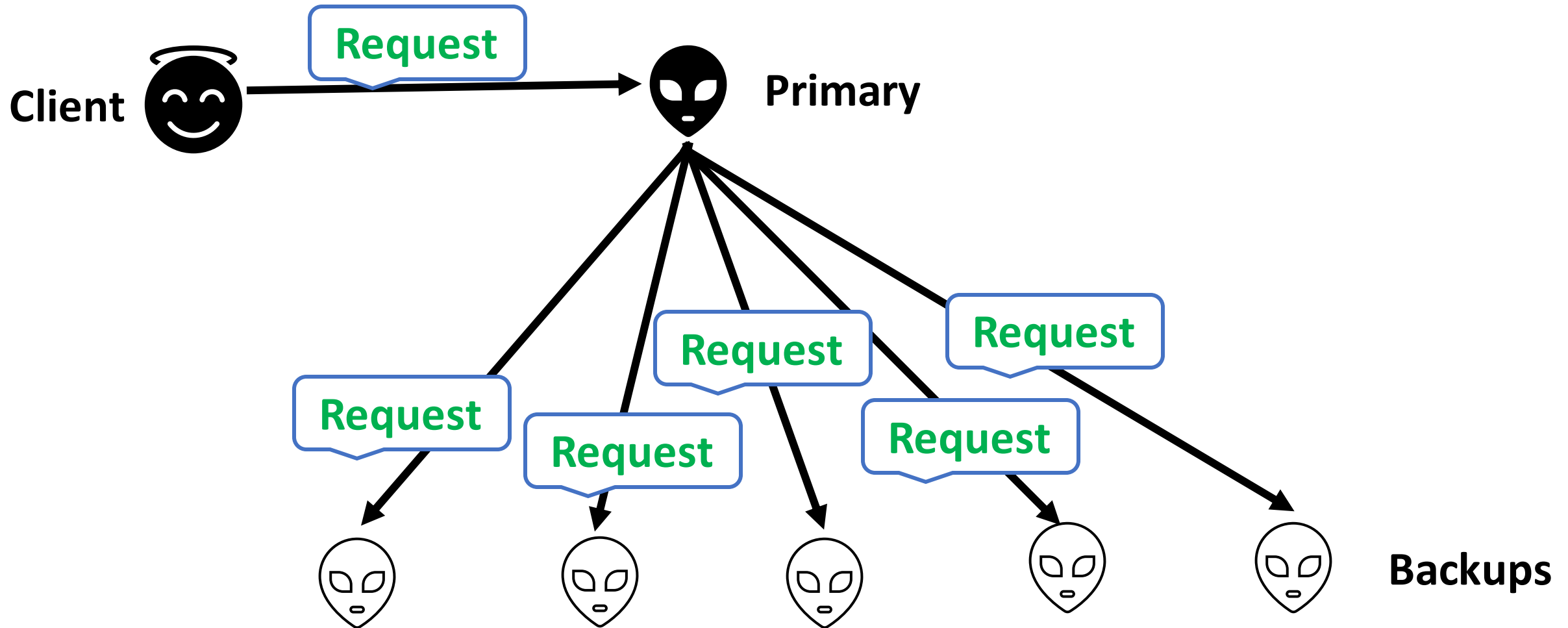
- Based on State Machine Replication
  - Considers  $3F + 1$  replicas where  $F$  can be the maximum number of faulty replicas
- The replicas move through a succession of configurations, known as ***views***
  - One replica in a view is considered as the **primary** (works like a leader), and others are considered **backups**
  - The primary proposes a value (similar to the Proposers in Paxos), and the backups accept the value (similar to the Paxos Acceptors)
  - When the primary is detected as faulty, the view is changed – PBFT elects a new primary and a new view is initiated
  - Every view is identified by a unique integer  $v$
  - Only the messages from the current view is accepted

# PBFT – Broad Idea

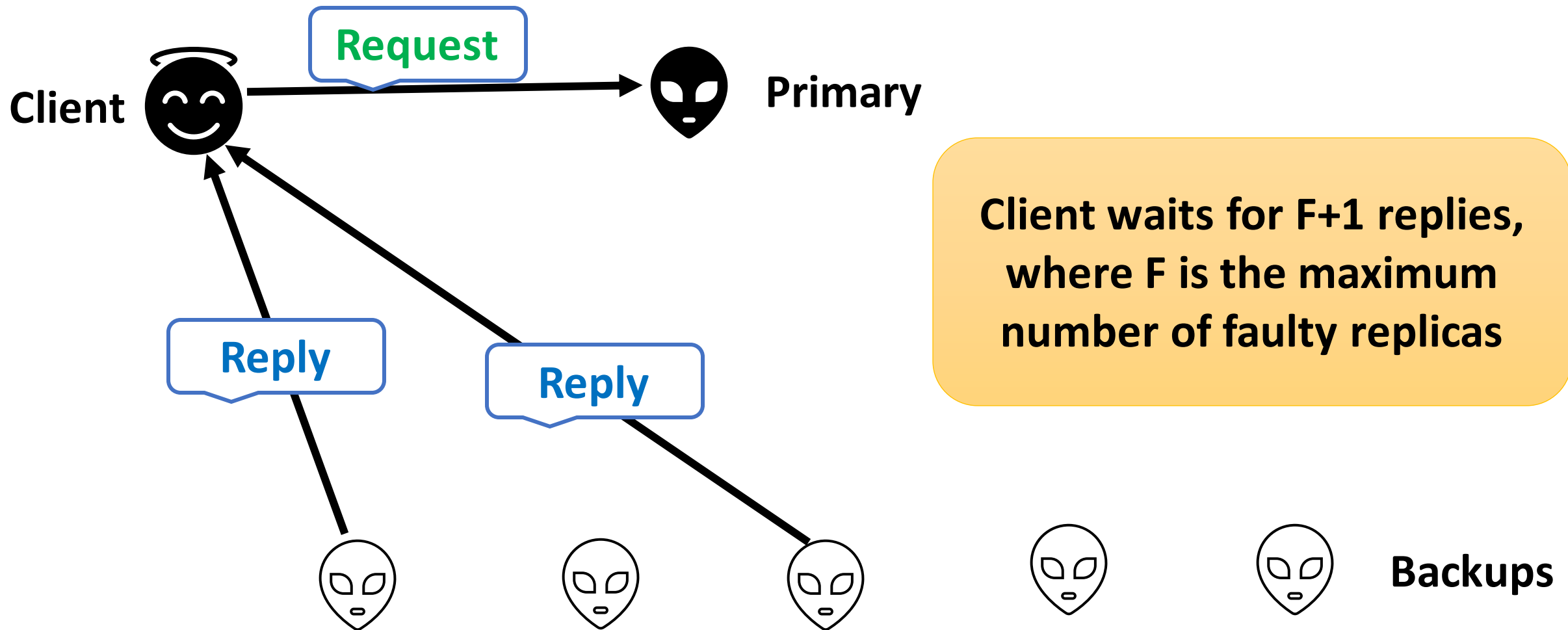




# PBFT – Broad Idea



# PBFT – Broad Idea

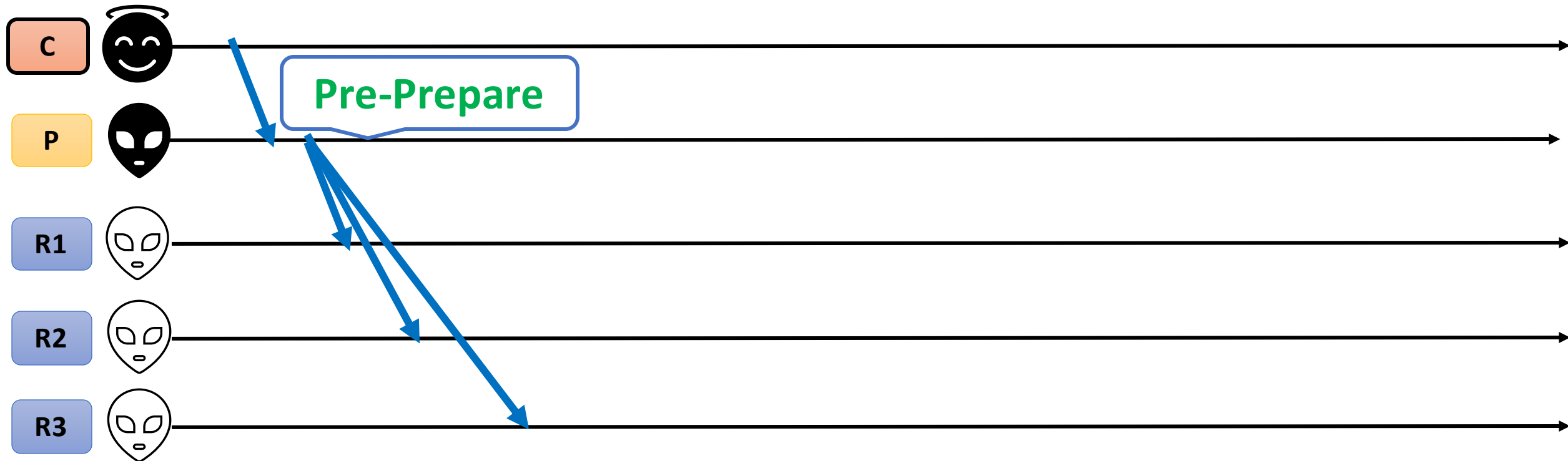


# PBFT – The Algorithm



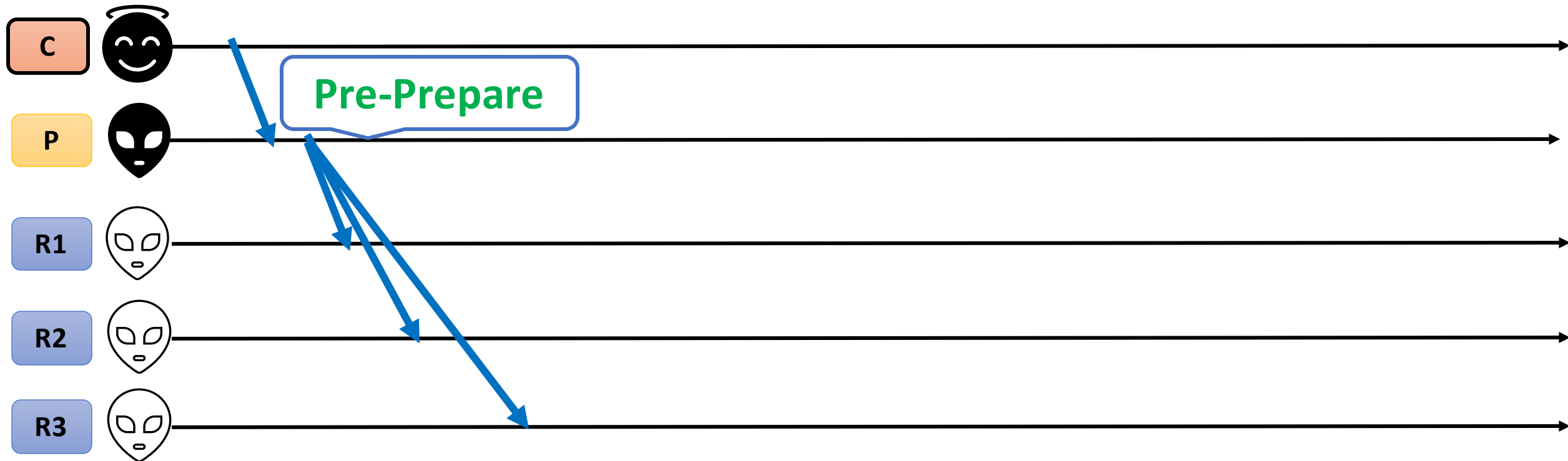
- The protocol starts by the client sending a Request message to the primary
- The primary collects all the Request messages from different clients and order them based on certain pre-defined logic

# PBFT – The Algorithm



- Primary assigns a sequence number  $n$  to the Request (or a set of Requests) and multicast a message  $\langle \langle \text{PRE-PREPARE}, v, n, d \rangle_{\beta_p}, m \rangle$  to all the backups
  - $v$  is the current view number,  $d$  is the message digest,  $m$  is the message
  - $\beta_p$  is the private key of the primary

# PBFT – The Algorithm



- Pre-prepare works as a proof that the Request was assigned a sequence number  $n$  for the view  $v$

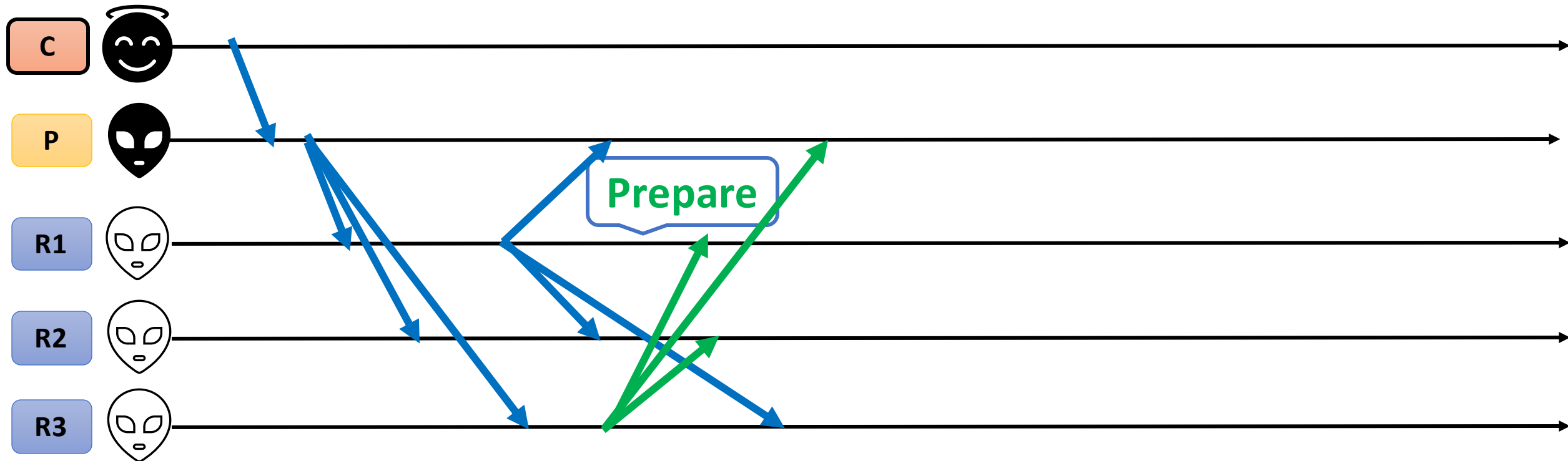
# PBFT – The Algorithm

A backup accepts the Pre-prepare message, if

- The signature is correct and  $d$  is the digest of the message  $m$
- The backup is in view  $v$
- It has not received a different Pre-Prepare message with sequence  $n$  and view  $v$  with a different message digest
- The sequence number is within a threshold (the message is not too old – prevents a reply attack)

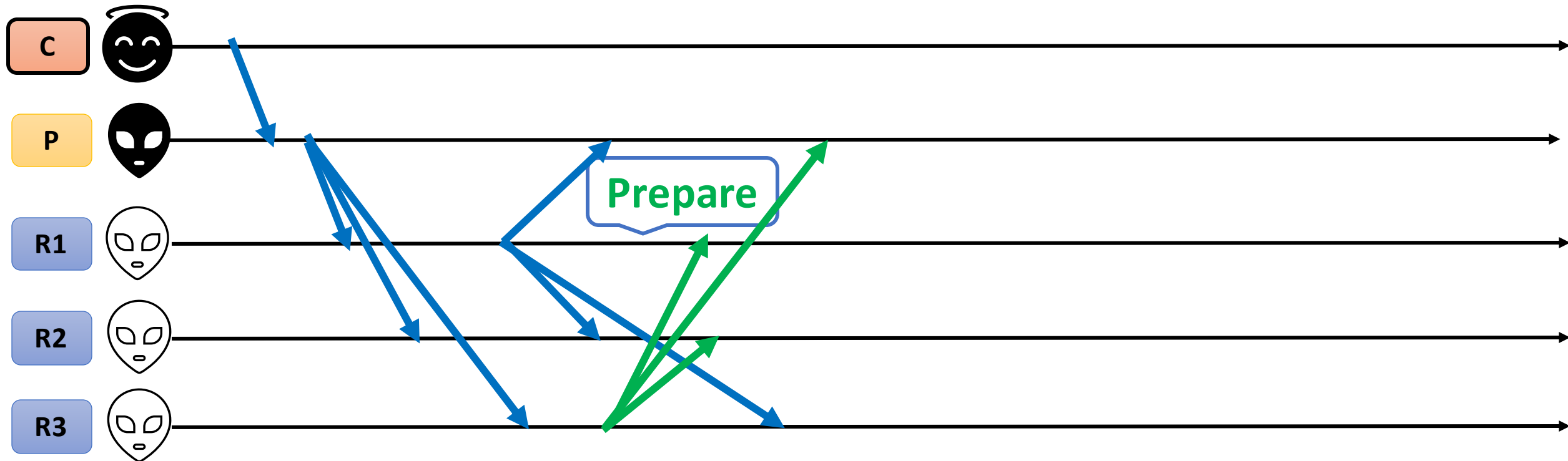


# PBFT – The Algorithm



- The correct backups send a Prepare message to all other backups including the primary – works as proof that the backups agree on the message with the sequence number  $n$  under view  $v$

# PBFT – The Algorithm



- Message format for backup  $k$  :  $\langle \text{PREPARE}, v, n, d, k \rangle_{\beta_k}$

# PBFT – The Algorithm

Primary and backups accepts the Prepare message, if

- The signatures are correct
- View number is equal to the current view
- Sequence number is within a threshold (note that messages may be received **out of order** – so a backup may receive the Prepare message before the corresponding Pre-prepare message – so it needs to keep track of all the messages received)

# PBFT – Three Phase Commit

- Pre-prepare and Prepare ensure that non-faulty replicas guarantee on a **total order** for the requests within a view

# PBFT – Three Phase Commit

- Pre-prepare and Prepare ensure that non-faulty replicas guarantee on a **total order** for the requests within a view
- Assumptions for Commit:
  - Primary is non-faulty
  - You may have a maximum of  $f$  faults including Crash + Network + Byzantine

# PBFT – Three Phase Commit

- A message is committed if
  - $2f$  Prepare from different backups matches with the corresponding Pre-prepare
  - You have total  $2f + 1$  votes (one from the primary that you already have!) from the non-faulty replicas

# PBFT – Three Phase Commit

- A message is committed if
  - $2f$  Prepare from different backups matches with the corresponding Pre-prepare
  - You have total  $2f + 1$  votes (one from the primary that you already have!) from the non-faulty replicas
- Note that all  $2f + 1$  votes may not be same
  - You have votes from Byzantine faulty replicas as well



# Why $2f + 1$ Votes? The idea of Quorum

- **Quorum:** Minimum number of votes a distributed transaction needs to obtained to get committed
  - Proposed by David Gifford in 1979 (Gifford, David K. (1979). *Weighted voting for replicated data*. SOSP '79)
  - Widely used in Commit protocols and Replica management

# Why $2f + 1$ Votes? The idea of Quorum

- **Quorum:** Minimum number of votes a distributed transaction needs to obtained to get committed
  - Proposed by David Gifford in 1979 (Gifford, David K. (1979). *Weighted voting for replicated data*. SOSP '79)
  - Widely used in Commit protocols and Replica management
- **Byzantine Dissemination Quorum:**
  - **Intersection:** Any two quorums have at least one correct replica in common
  - **Availability:** There is always a quorum available with no faulty replicas

# Why $2f + 1$ Votes? The idea of Quorum

- **Quorum:** Minimum number of votes a distributed transaction needs to obtained to get committed
  - Proposed by David Gifford in 1979 (Gifford, David K. (1979). *Weighted voting for replicated data*. SOSP '79)
  - Widely used in Commit protocols and Replica management
- **Byzantine Dissemination Quorum:**
  - **Intersection:** Any two quorums have at least one correct replica in common
  - **Availability:** There is always a quorum available with no faulty replicas
- PBFT uses Byzantine Dissemination Quorum with  $2f + 1$  replicas

# Quorum in PBFT

- You have  $f$  number of faulty nodes – you need at least  **$3f + 1$**  replicas to reach consensus
  - But you do not know whether those are Crash faults, Network faults, or Byzantine Faults

# Quorum in PBFT

- You have  $f$  number of faulty nodes – you need at least  $3f + 1$  replicas to reach consensus
  - But you do not know whether those are Crash faults, Network faults, or Byzantine Faults
- **Case 1: All  $f$  are Crash or Network faulty – You'll not receive messages from them!**
  - You'll receive  $2f + 1$  Prepare messages from non-faulty nodes
  - All these  $2f + 1$  are non-faulty votes – you can reach to an agreement

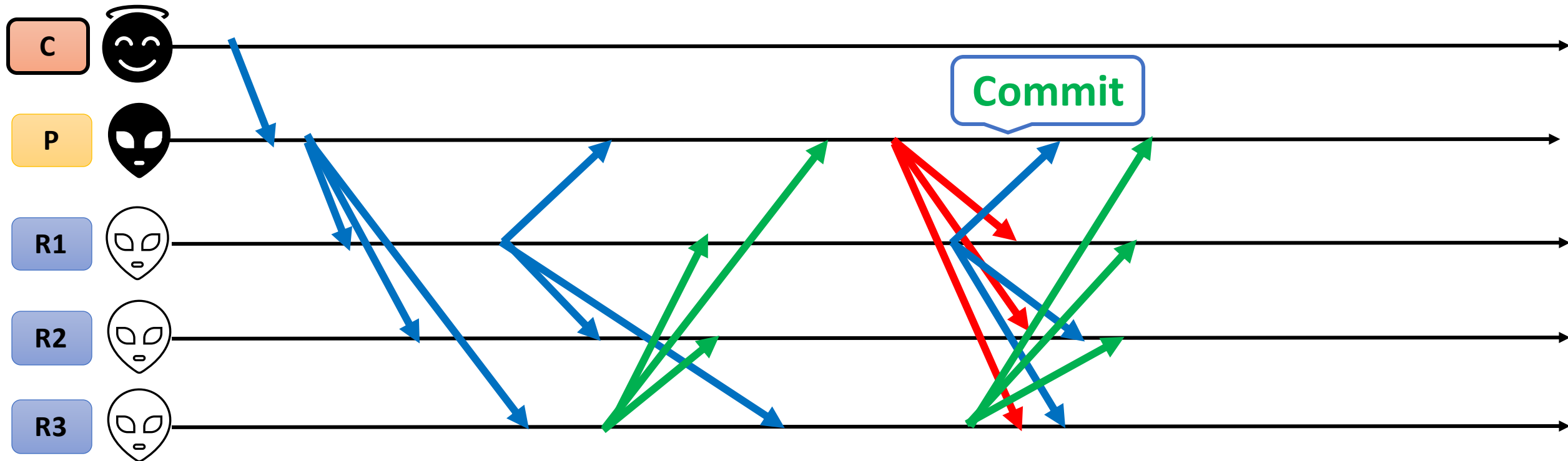
# Quorum in PBFT

- You have  $f$  number of faulty nodes – you need at least  $3f + 1$  replicas to reach consensus
  - But you do not know whether those are Crash faults, Network faults, or Byzantine Faults
- **Case 2: All  $f$  are Byzantine faulty – they send messages!**
  - You may receive at most  $3f + 1$  Prepare messages (votes) --  $f$  are from Byzantine nodes
  - Sufficient to wait till  $2f + 1$  Prepare messages – even if  $f$  are faulty, you still have  $f+1$  non-faulty votes
  - You cannot wait for  $f+1$ , the first  $f$  might be all faulty

# Quorum in PBFT

- You have  $f$  number of faulty nodes – you need atleast  $3f + 1$  replicas to reach consensus
  - But you are on an **asynchronous channel** – messages get delayed and can be received out of order
- **Case 2: A**
  - You must wait until you receive  $2f + 1$  Prepare messages – once you received  $2f + 1$  votes, you can safely take a decision based on majority voting
  - Sufficiently many replicas must have  $f$  replicas
  - You cannot wait for  $f + 1$ , the first  $f$  might be all faulty

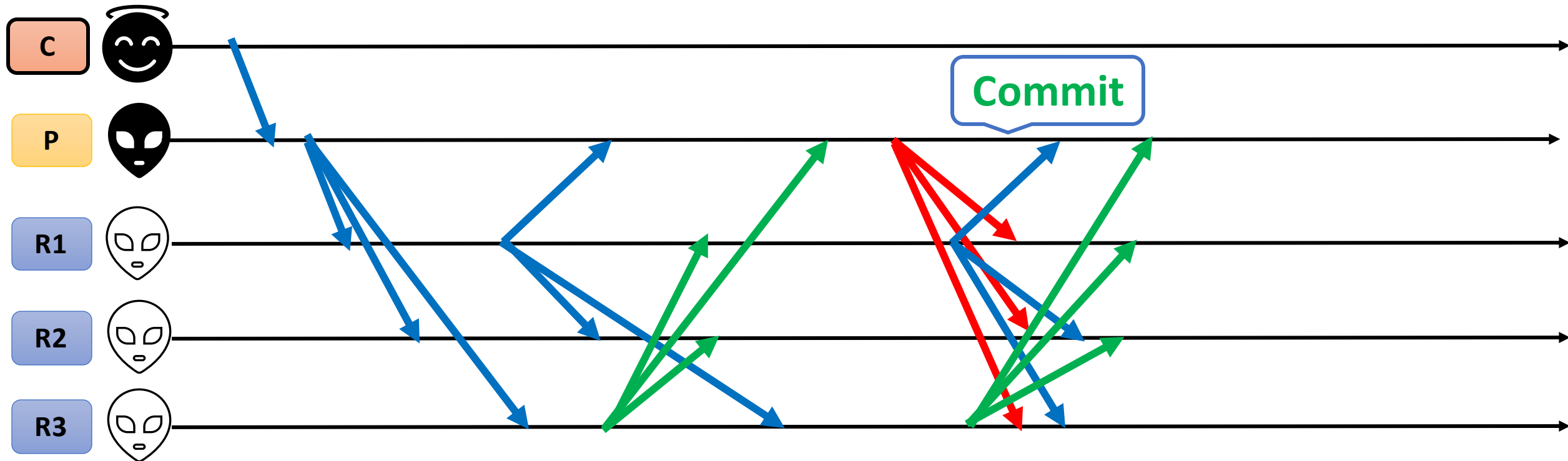
# PBFT – The Algorithm



- Message format for replica  $k$  :  $\langle \text{COMMIT}, v, n, d, k \rangle_{\beta_k}$

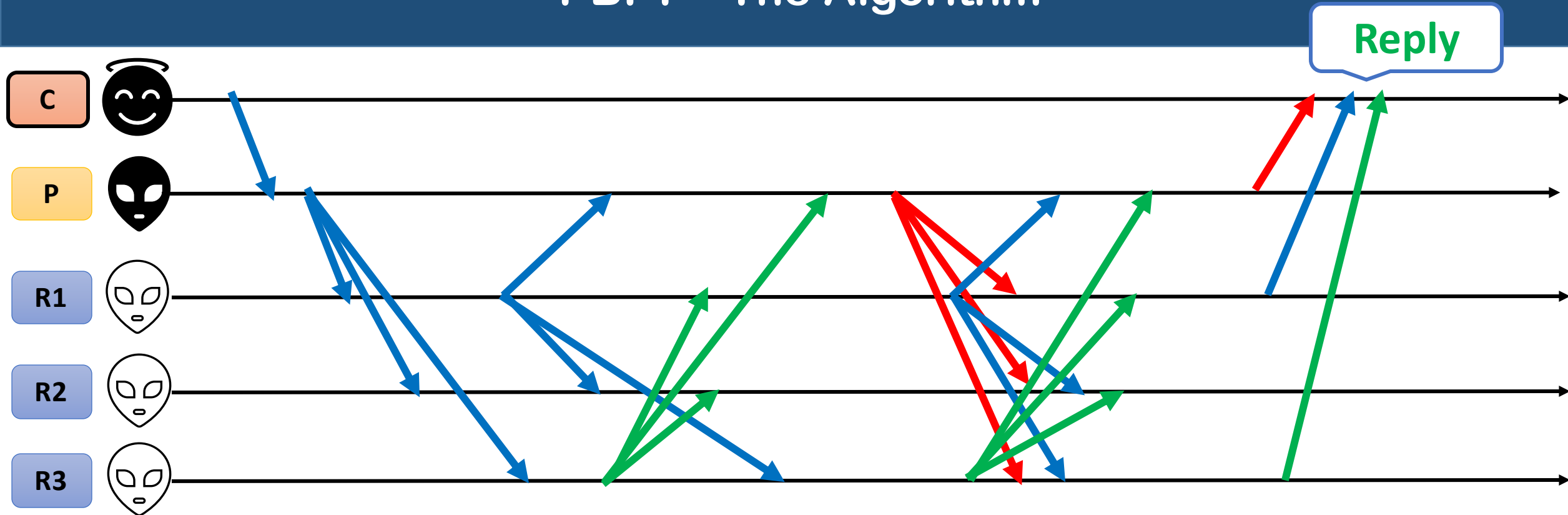


# PBFT – The Algorithm



- Message format for replica  $k$  :  $\langle \text{COMMIT}, v, n, d, k \rangle_{\beta_k}$
- The protocol is committed for a replica when
  - It has sent the Commit message
  - It has received  $2f$  Commit messages from other replicas

# PBFT – The Algorithm



- Message format for replica  $k$  :  $\langle \text{COMMIT}, v, n, d, k \rangle_{\beta_k}$
- The protocol is committed for a replica when
  - It has sent the Commit message
  - It has received  $2f$  Commit messages from other replicas

# Liveness and Weak Synchrony

- Unlike multiple Paxos proposers, PBFT works with a single Primary
  - Ping-pong does not arise from the proposals from multiple replicas
  - However, a replica needs to wait for  $2f + 1$  votes (Prepare and Commit messages)
- However, a primary may fail – the liveness gets hampered as the protocol cannot progress any further
  - Primary failure cannot be handled in a pure asynchronous system – you do not know whether it is a message delay from the primary, or a primary failure

# Liveness and Weak Synchrony

- Unlike multiple Paxos proposers, PBFT works with a single Primary
  - Ping-pong does not arise from the proposals from multiple replicas
  - However, a replica needs to wait for  $2f + 1$  votes (Prepare and Commit messages)
- However, a primary may fail – the liveness gets hampered as the protocol cannot progress any further
  - Primary failure cannot be handled in a pure asynchronous system – you do not know whether it is a message delay from the primary, or a primary failure
- **Weak Synchrony:** (1) Both sender and the receiver is correct, (2) Sender keeps retransmitting the messages until it is received, (3) There is an asymptotic upper bound on the message transmission delay

# View Change

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

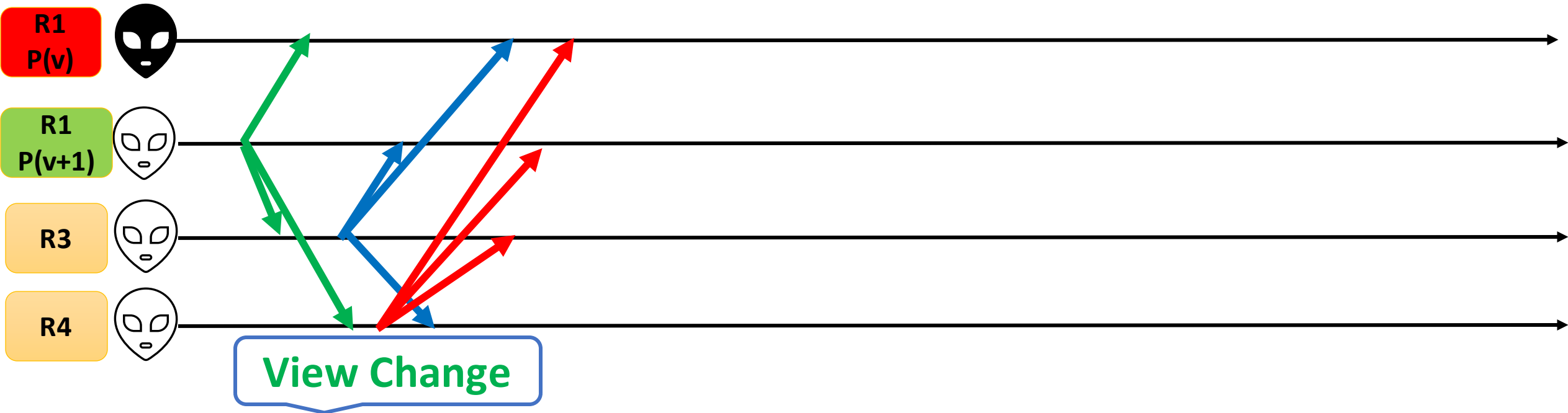
# View Change

- What if the **primary** is **faulty** ?
  - Non-faulty replicas detect the fault
  - Replicas together start view change operation
- View-change protocol provides **eventual liveness** -- Allows the system to make progress when primary fails
- If the primary fails, backups will not receive any message or will receive faulty messages from the primary
- View changes are triggered by timeouts (weak synchrony assumption)
  - 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 to the view  $v + 1$

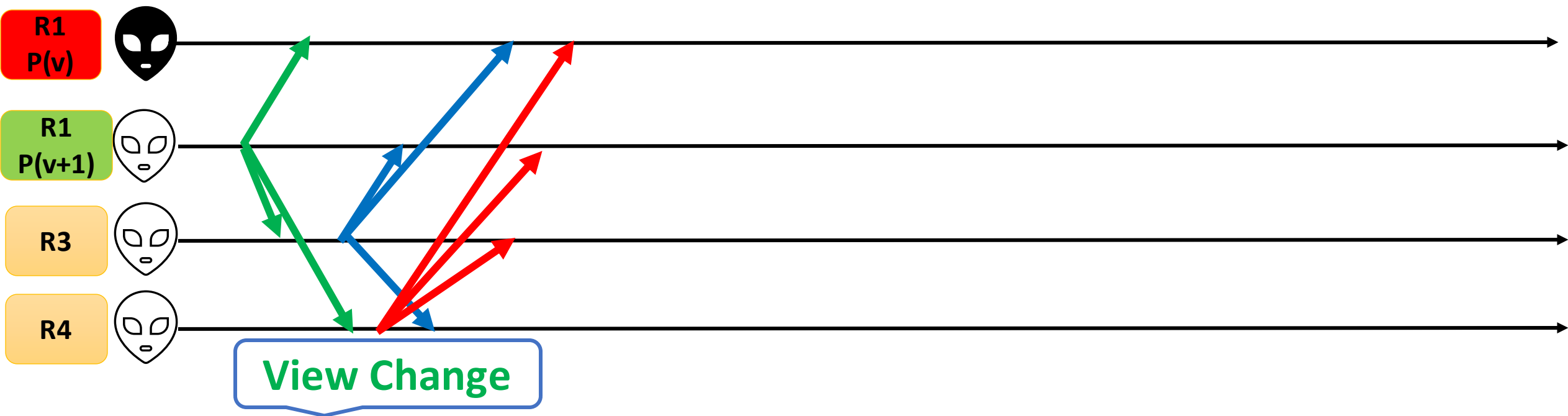
# The View Change Protocol



- Multicast the view change message  $\langle \text{VIEW-CHANGE}, v+1, n, C, P, k \rangle_{\beta_k}$ 
  - $n$  is the sequence number of last stable checkpoint  $s$  known to  $k$
  - $C$  is a set of  $2f + 1$  valid checkpoint messages corresponding to  $s$
  - $P$  is a set containing a set  $P_m$  for each request  $m$  that prepared at  $k$  with a sequence number higher than  $n$

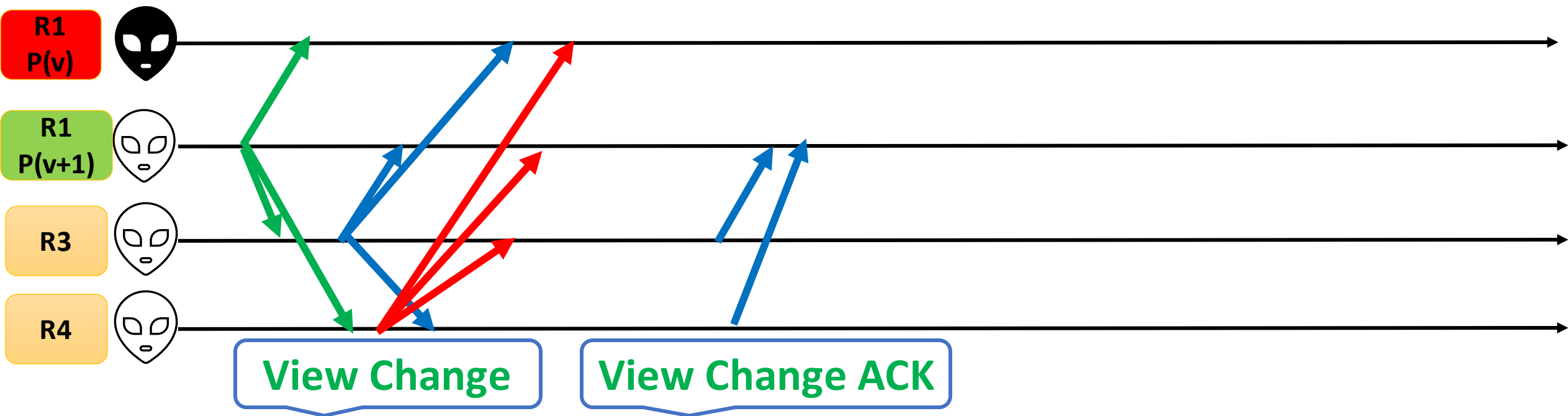


# The View Change Protocol



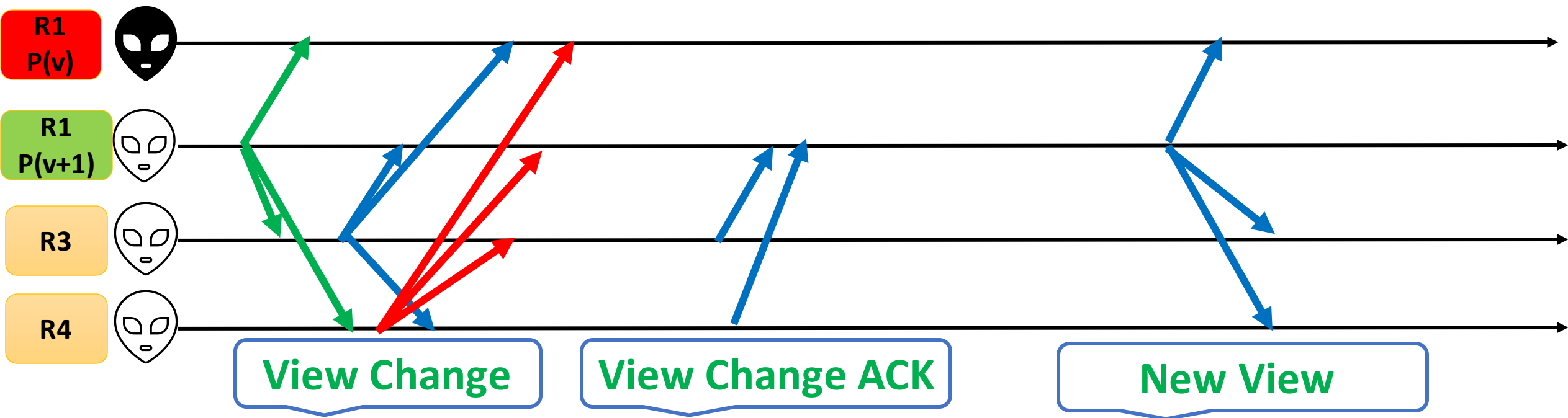
- The new view is initiated after receiving  $2f + 1$  View Change messages
- The primary of a view  $v$  is a replica  $p$  such that  $p = v \bmod |R|$ , where  $R$  is the set of replicas ( $|R| = 3f + 1$ , in general)

# The View Change Protocol



- Replicas send a View Change ACK – quorum is formed on these messages

# The View Change Protocol



- Replicas send a View Change ACK – quorum is formed on these messages
- New View message to initiate a new view

# The View Change Protocol

- The New View message is formatted as  $\langle \text{NEW-VIEW}, v+1, V, O \rangle_{\beta\_p}$ 
  - $V$  is the set containing the valid view change messages received by the new primary plus the view change message that it sent (**or would have been sent**)
  - $O$  is the set of pre-prepare messages

# The View Change Protocol

- How do the new primary computes O?
  - The primary determines the sequence number *min-s* of the latest stable checkpoint in *V* and the highest sequence number *max-s* in a prepare message in *V*.
  - The primary creates a new pre-prepare message for view *v+1* for each sequence number between *min-s* and *max-s*.
  - Two cases are handled
    - There is at least one set in the *P* component of some view-change message in *V* with sequence number *n*: creates a new message  $\langle \text{PRE-PREPARE}, v+1, n, d \rangle_{\beta\_p}$ , *d* is the request digest in the pre-prepare message for sequence number *n* with the highest view number in *V*.
    - There is no such set: creates a new message  $\langle \text{PRE-PREPARE}, v+1, n, d^{\text{null}} \rangle_{\beta\_p}$ ,  $d^{\text{null}}$  is the digest of a special null request; a null request goes through the protocol like other requests, but its execution is a no-op.

# The View Change Protocol

- The New View message is formatted as  $\langle \text{NEW-VIEW}, v+1, V, O \rangle_{\beta_p}$ 
  - $V$  is the set containing the valid view change messages received by the new primary plus the view change message that it sent (**or would have been sent**)
  - $O$  is the set of pre-prepare messages
- Next the primary appends the messages into its own log.
  - If  $\text{min-s}$  is greater than the sequence number of its latest stable checkpoint, the primary also inserts the proof of stability for the checkpoint with sequence number  $\text{min-s}$  in its log
  - It then enters the view  $v+1$  and starts accepting messages for the view  $v+1$
- Backups accept the view change message for view  $v+1$  if it is signed properly, if the view-change messages it contains are valid for view 1, and if the set  $O$  is correct (checked through a similar computation as of the primary).

# Garbage Collection

- When do a replica can discard the messages it received from others?
  - For the safety condition to hold, messages must be kept in a replica's log until it knows that
    - The requests they concern have been executed by at least  $f+1$  non-faulty replicas
    - It can prove this to others in view changes.
  - If some replica misses messages that were discarded by all other non-faulty replicas, it will need to be brought up to date by transferring all or a portion of the service state.
    - Therefore, replicas also need some proof that the state is correct.

# Garbage Collection

- Generating the proofs after every operation is expensive -> PBFT replicas generate these proofs at every  $i^{\text{th}}$  sequence number (say  $i=100$ )
  - Are called the checkpoints: Checkpoints with a proof are called **stable checkpoints**
- A replica maintains several service states
  - Last stable checkpoint
  - Zero or more checkpoints which are not stable
  - The current state
- Copy on Write to maintain the states – space efficient



# How Does a Replica Generate the Proof of a Checkpoint

- When a replica  $i$  produces a checkpoint, it multicasts a message  $\langle \text{CHECKPOINT}, n, d, i \rangle_{\beta_i}$  to other replicas
  - $n$  is the sequence number of the last request whose execution is reflected in the state and  $d$  is the digest of the state
- Each replica collects checkpoint messages in its log until it has  $2f+1$  of them for the sequence number  $n$  with the same digest  $d$  signed by different replicas (including possibly its own such message).
  - These messages are the proof of correctness for the checkpoint
- Once a checkpoint with sequence number  $n$  becomes stable, a replica deletes all old checkpoints for sequence number less than  $n$

## Another Logical Proof for $n = 3f + 1$

- You have at most  $f$  number of Byzantine faulty nodes in the system
  - The protocol must work after communicating with  $n-f$  number of replicas
- Say, you do not receive the responses from  $f$  number of replicas
  - These  $f$  replicas can be the ones which are faulty (as Byzantine faulty replicas can stop responding)
  - It might also be possible that the responses are delayed from these replicas as the communication channel is asynchronous; but these are actually the non-faulty replicas
    - So, the  $n-f$  responses that are received might have the responses from  $f$  faulty nodes
- For the second case, we need to ensure that the responses still have enough number of replicas from the non-faulty set
  - So,  $n - 2f > f \Rightarrow n > 3f$

## Must Read

- Gabriel Bracha and Sam Toueg. 1985. Asynchronous consensus and broadcast protocols. J. ACM 32, 4 (Oct. 1985), 824–840.  
<https://doi.org/10.1145/4221.214134>
- Lamport, Leslie, Robert Shostak, and Marshall Pease. "The Byzantine generals problem." Concurrency: the works of Leslie Lamport. 2019. 203-226.
- Castro, Miguel, and Barbara Liskov. "Practical byzantine fault tolerance and proactive recovery." ACM Transactions on Computer Systems (TOCS) 20.4 (2002): 398-461.

