Byzantine Agreement Protocols

Department of Computer Science and Engineering

INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR

Sandip Chakraborty
sandipc@cse.iitkgp.ac.in

Byzantine Generals Problem

Lieutenant - 1

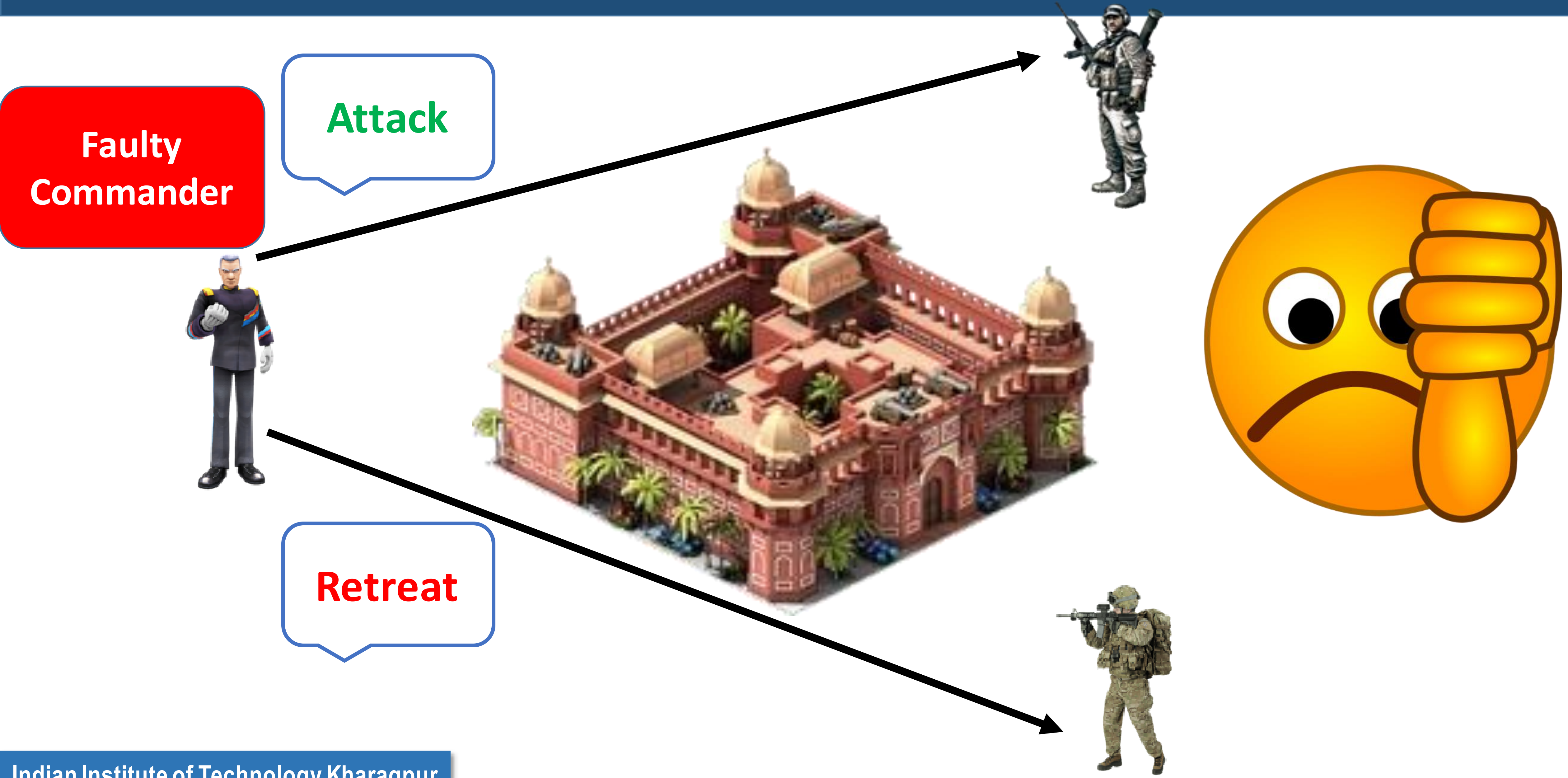
Click to add text

Commander

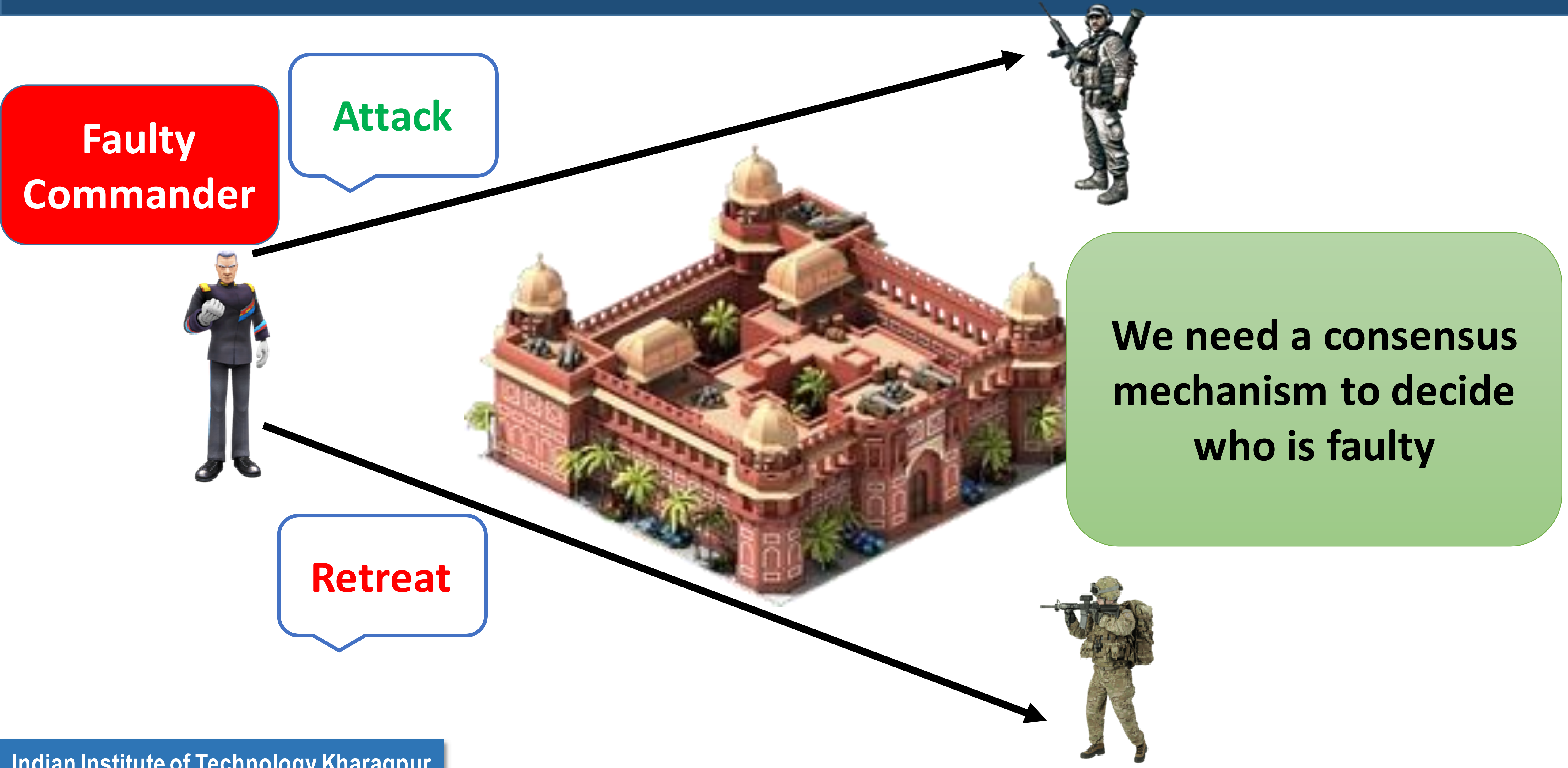
Lieutenant - 2

Byzantine Generals Problem

Byzantine Generals Problem



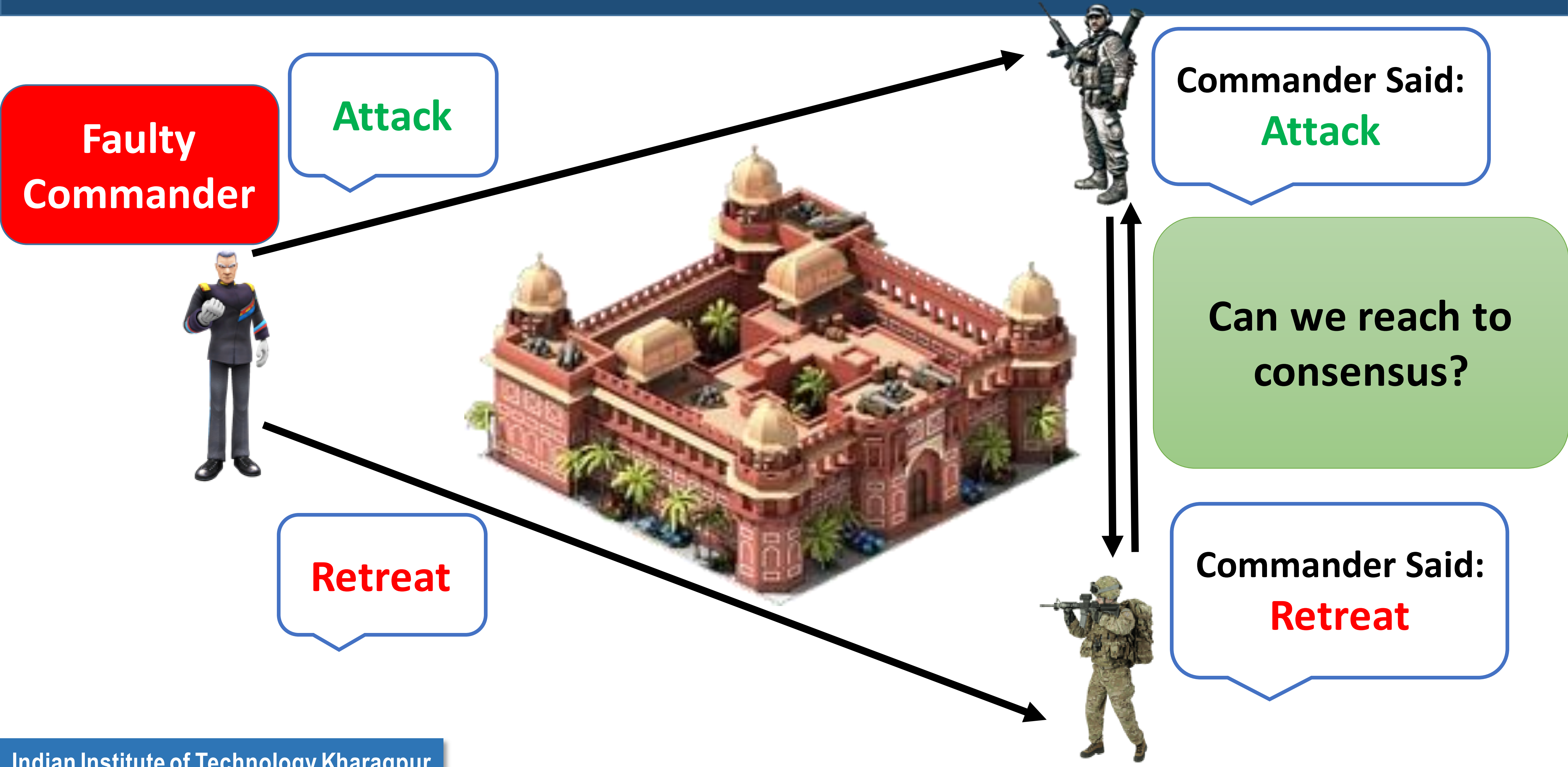
Byzantine Generals Problem



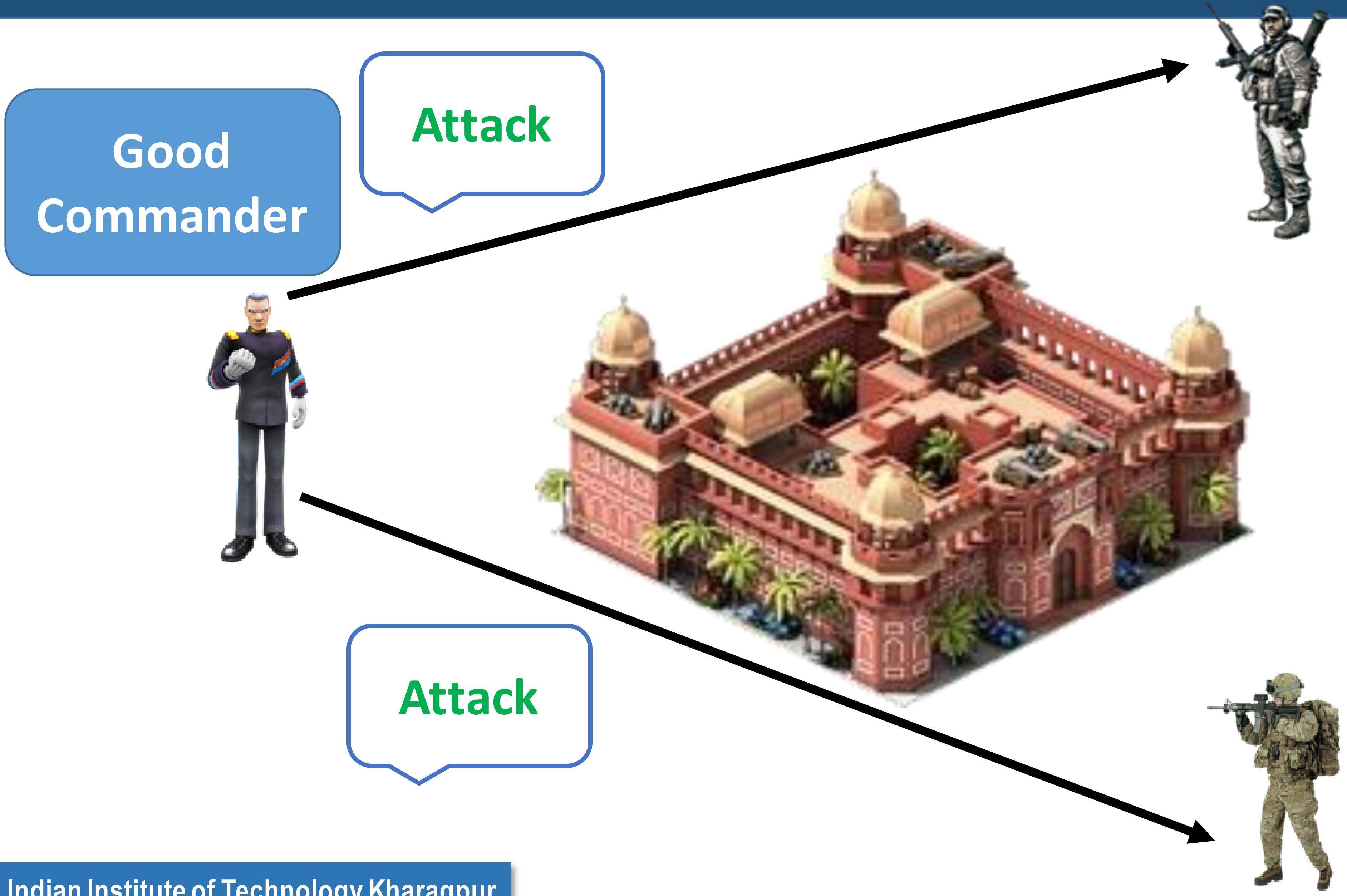
Byzantine Generals Problem

Byzantine Generals Problem

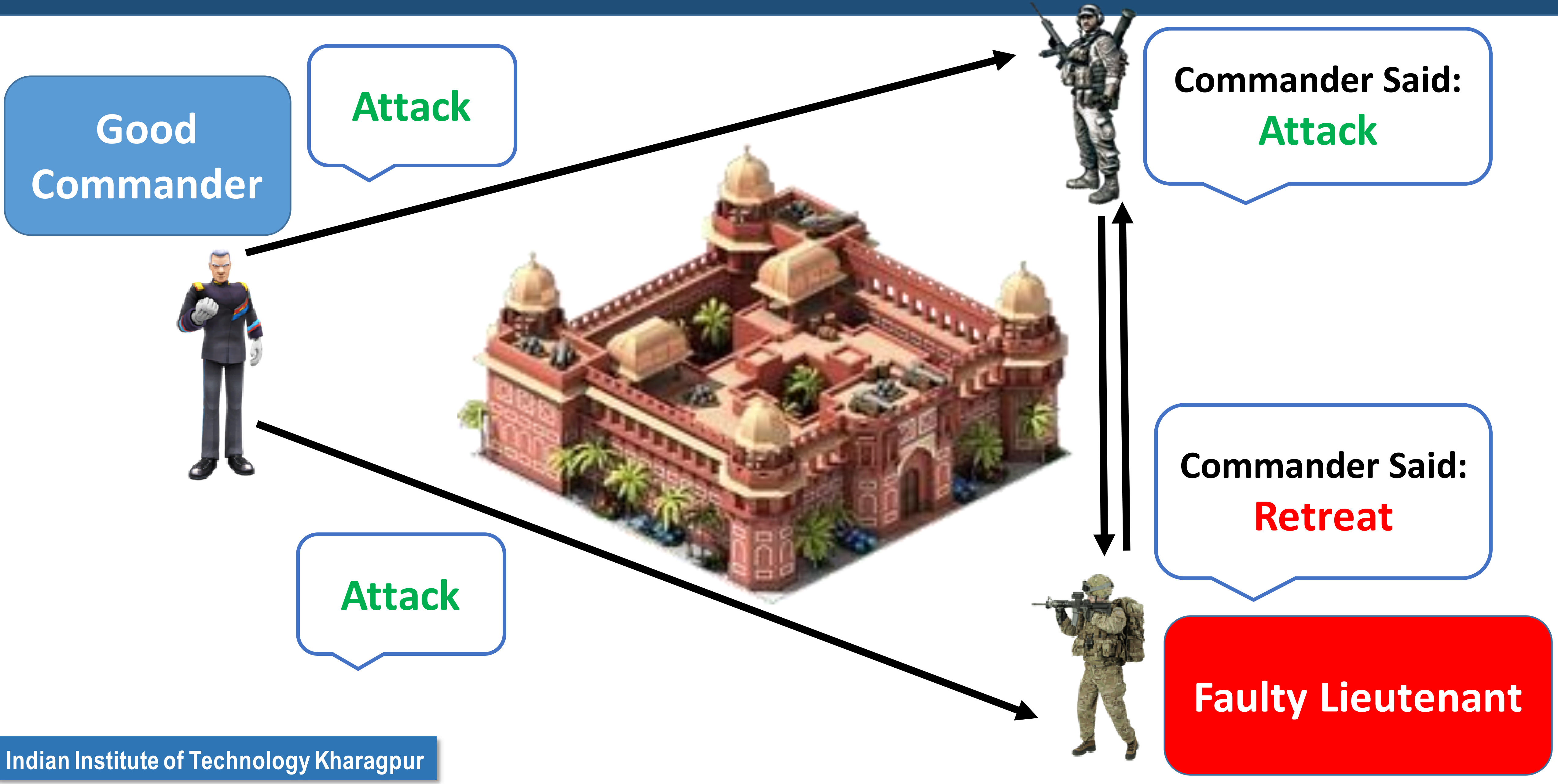
Byzantine Generals Problem



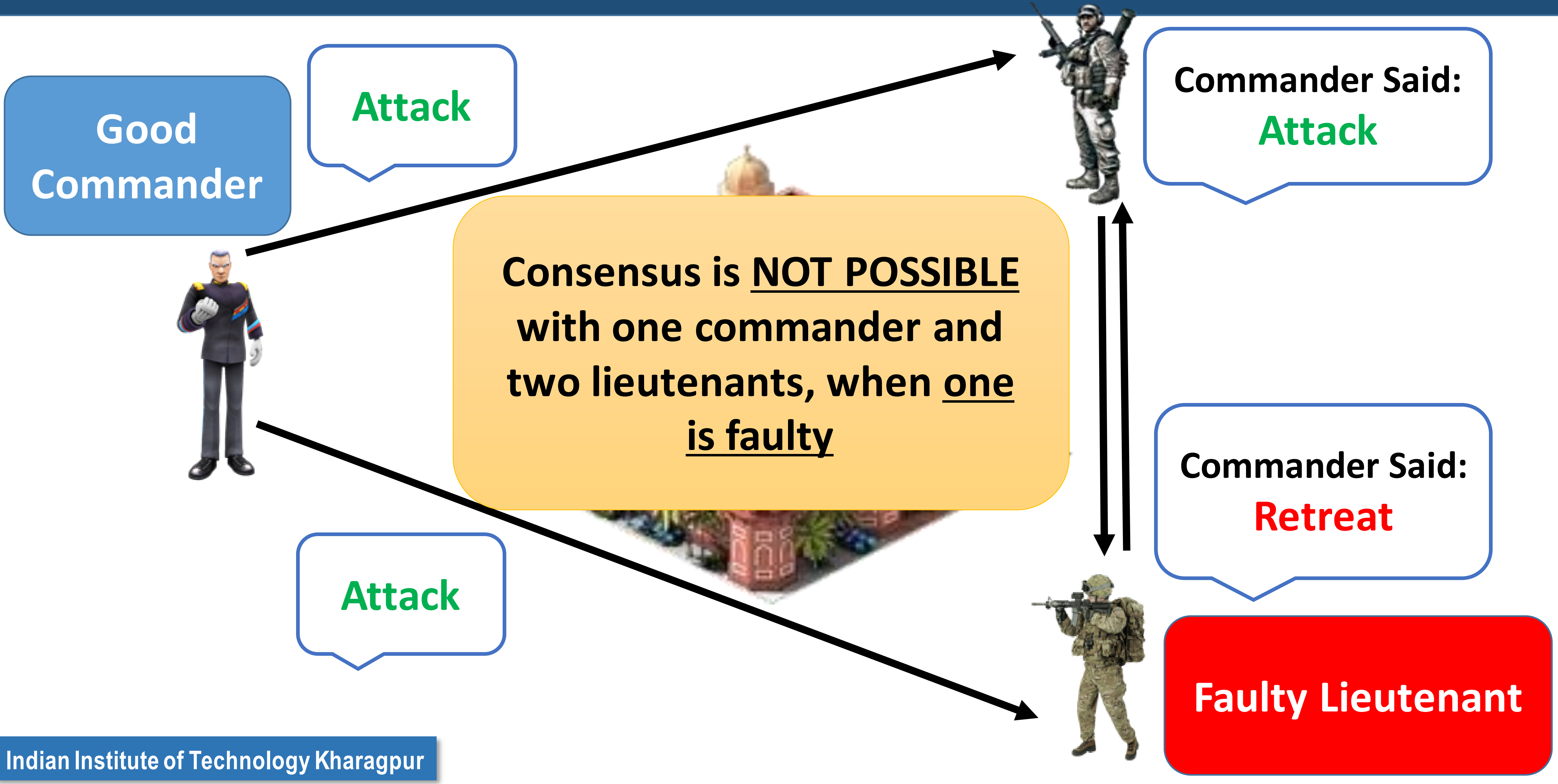
Byzantine Generals Problem



Byzantine Generals Problem

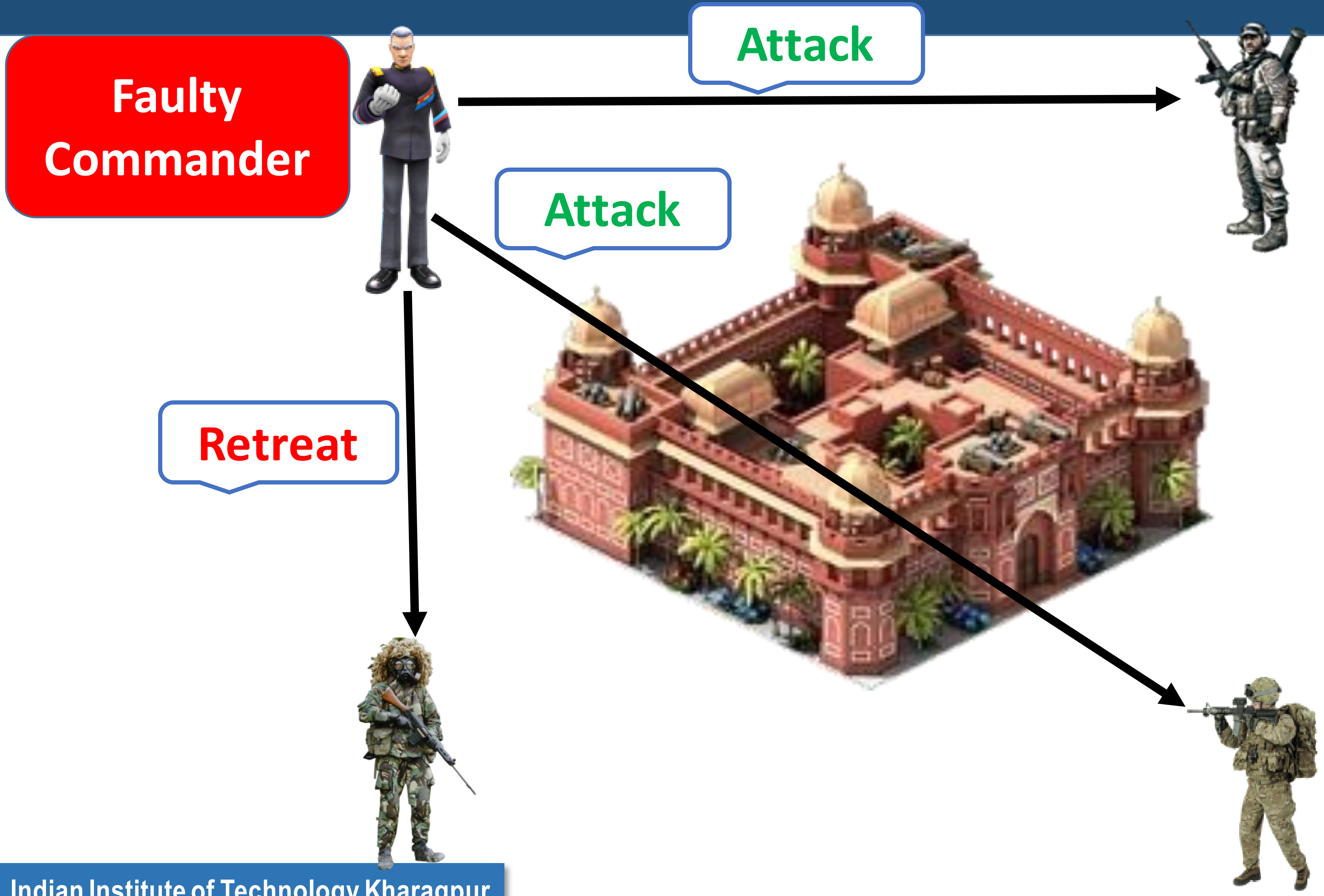


Byzantine Generals Problem

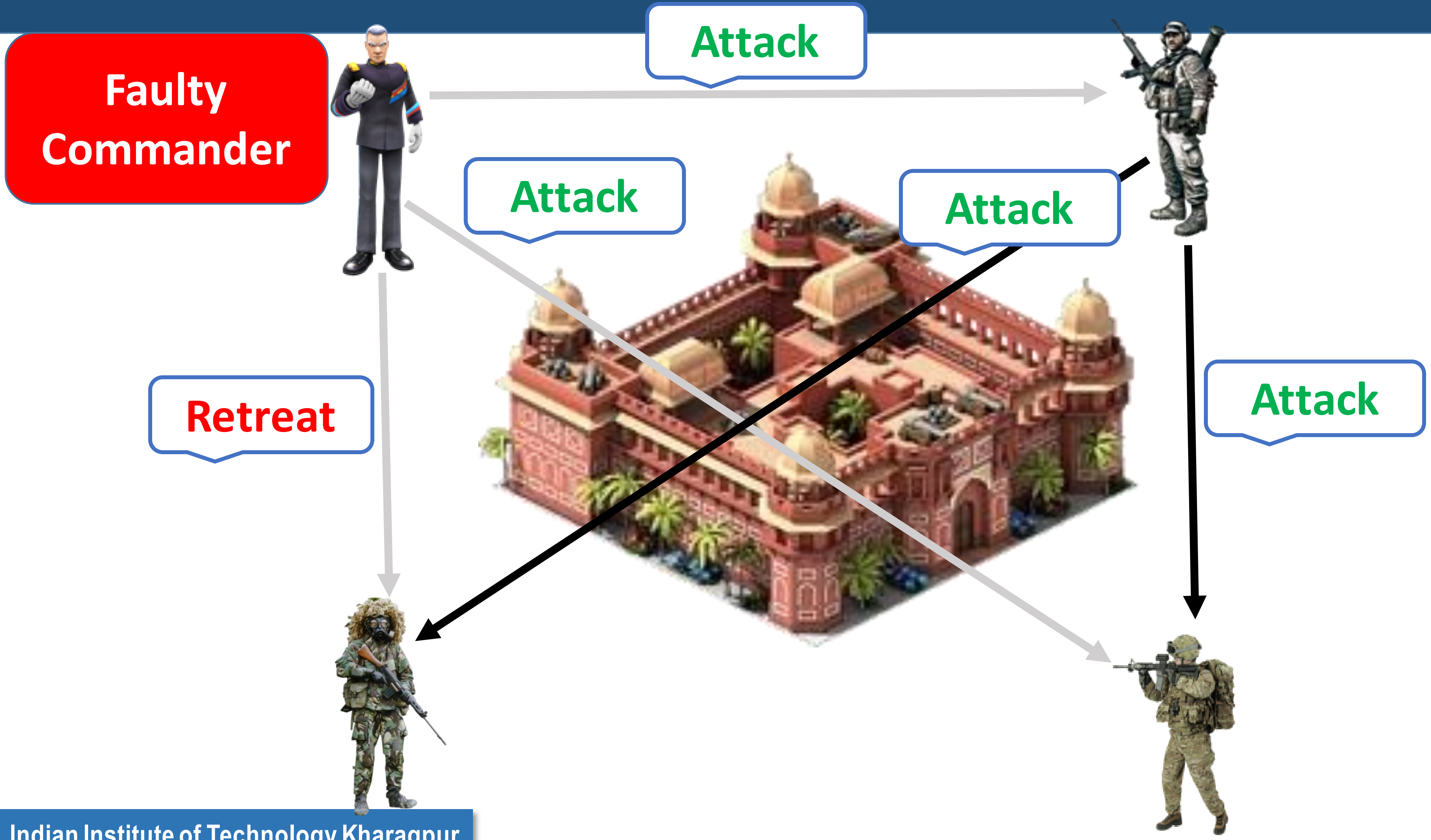


Byzantine Generals Problem – Three Lieutenants

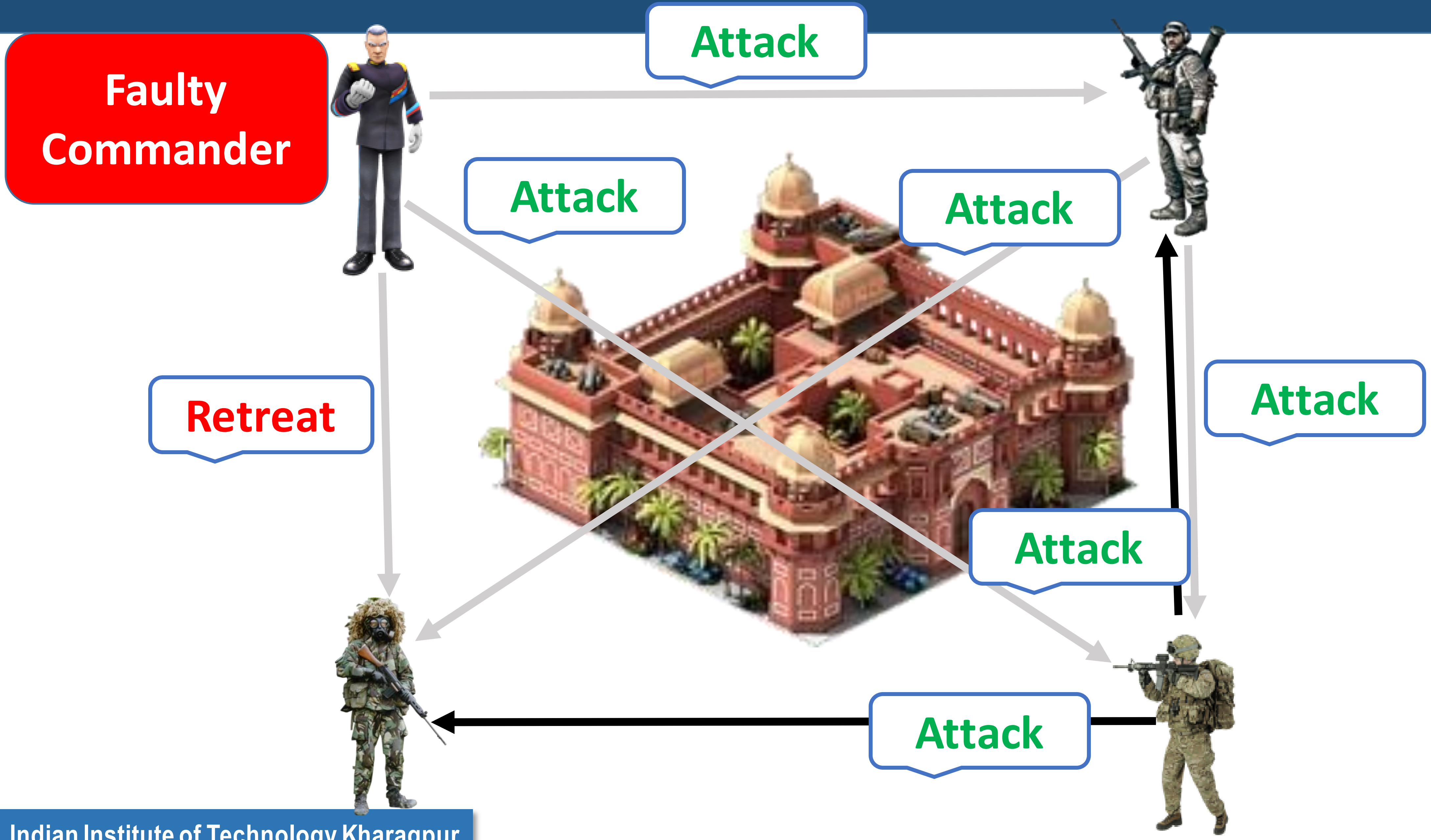
Byzantine Generals Problem – Three Lieutenants



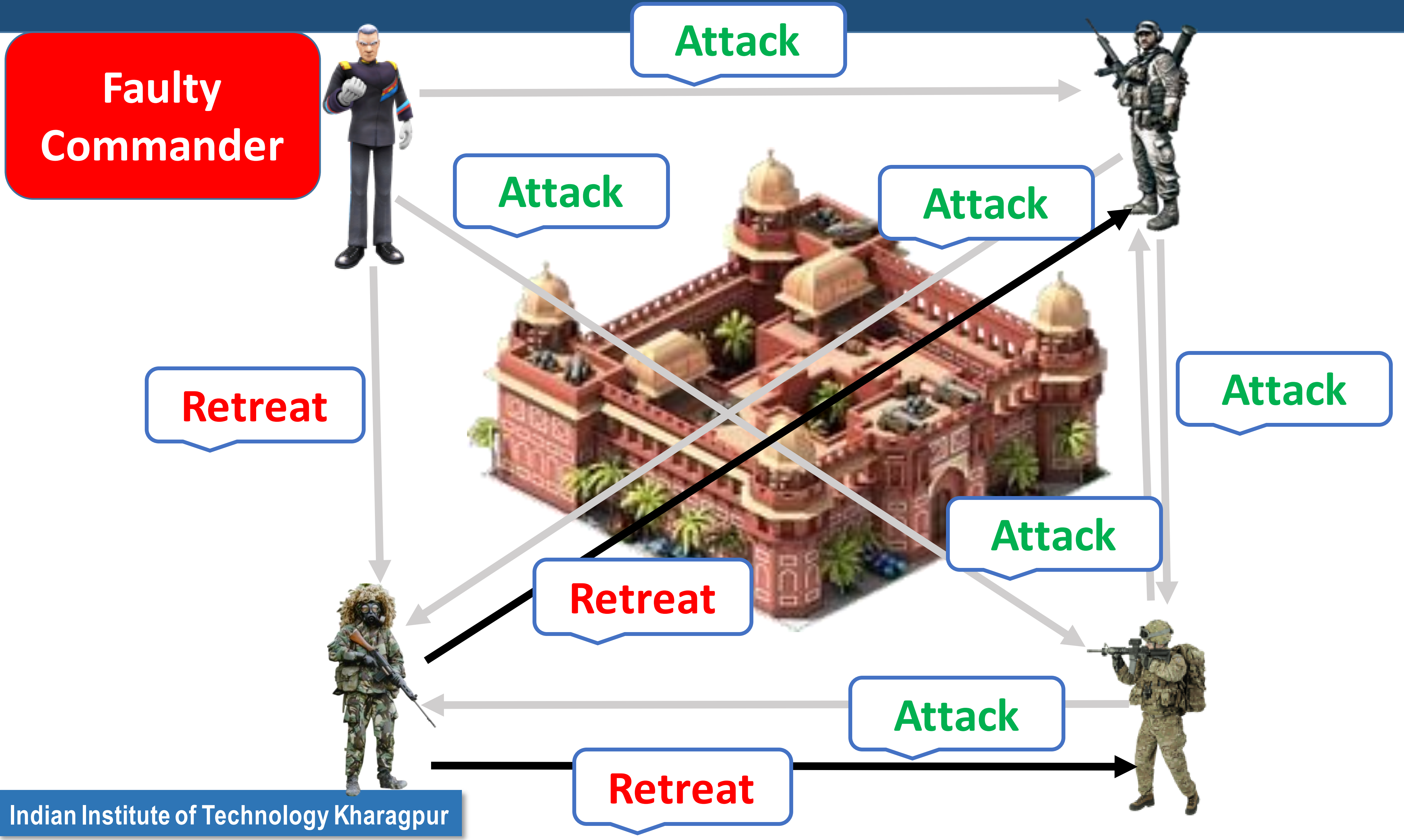
Byzantine Generals Problem – Three Lieutenants



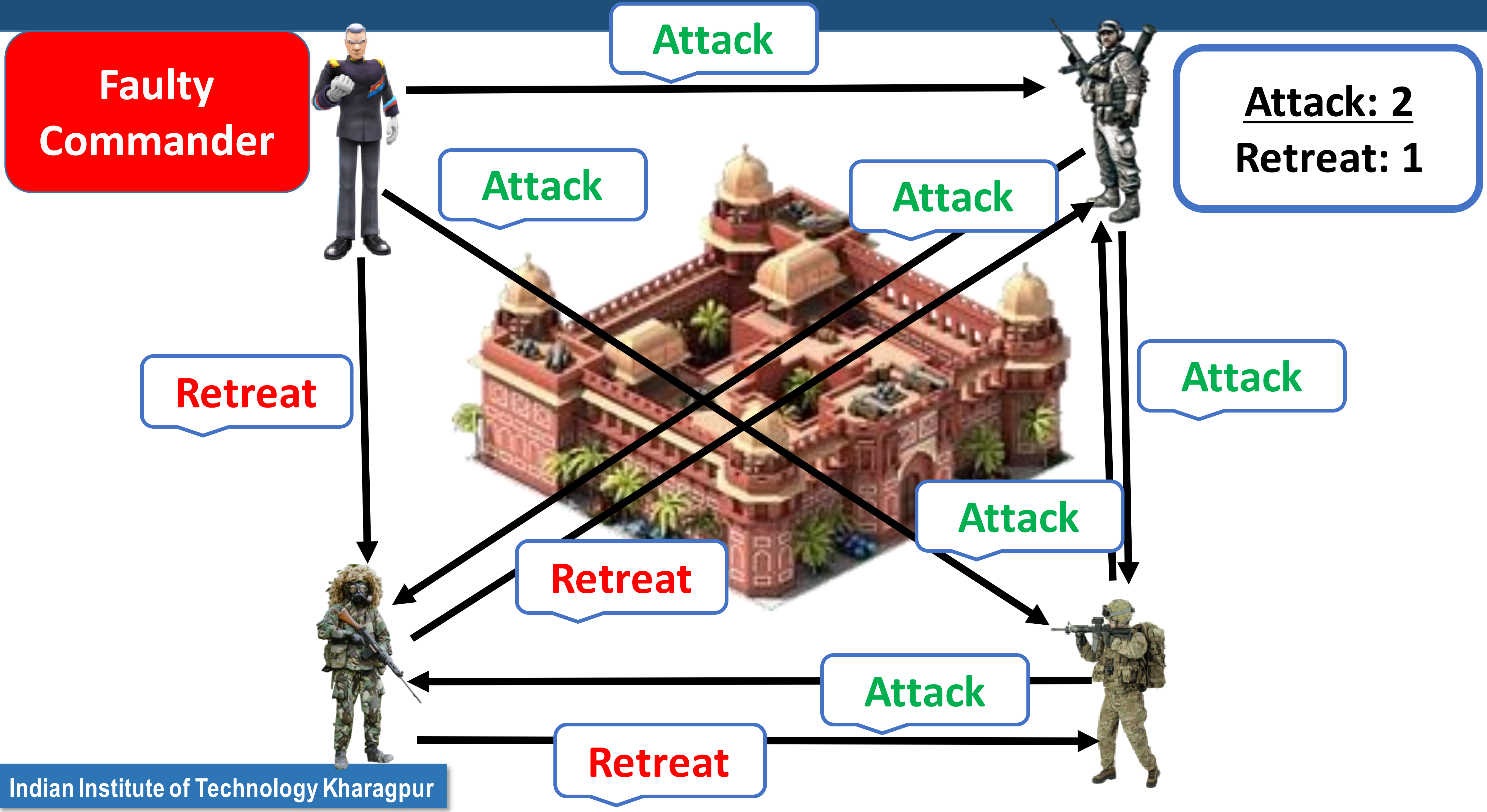
Byzantine Generals Problem – Three Lieutenants



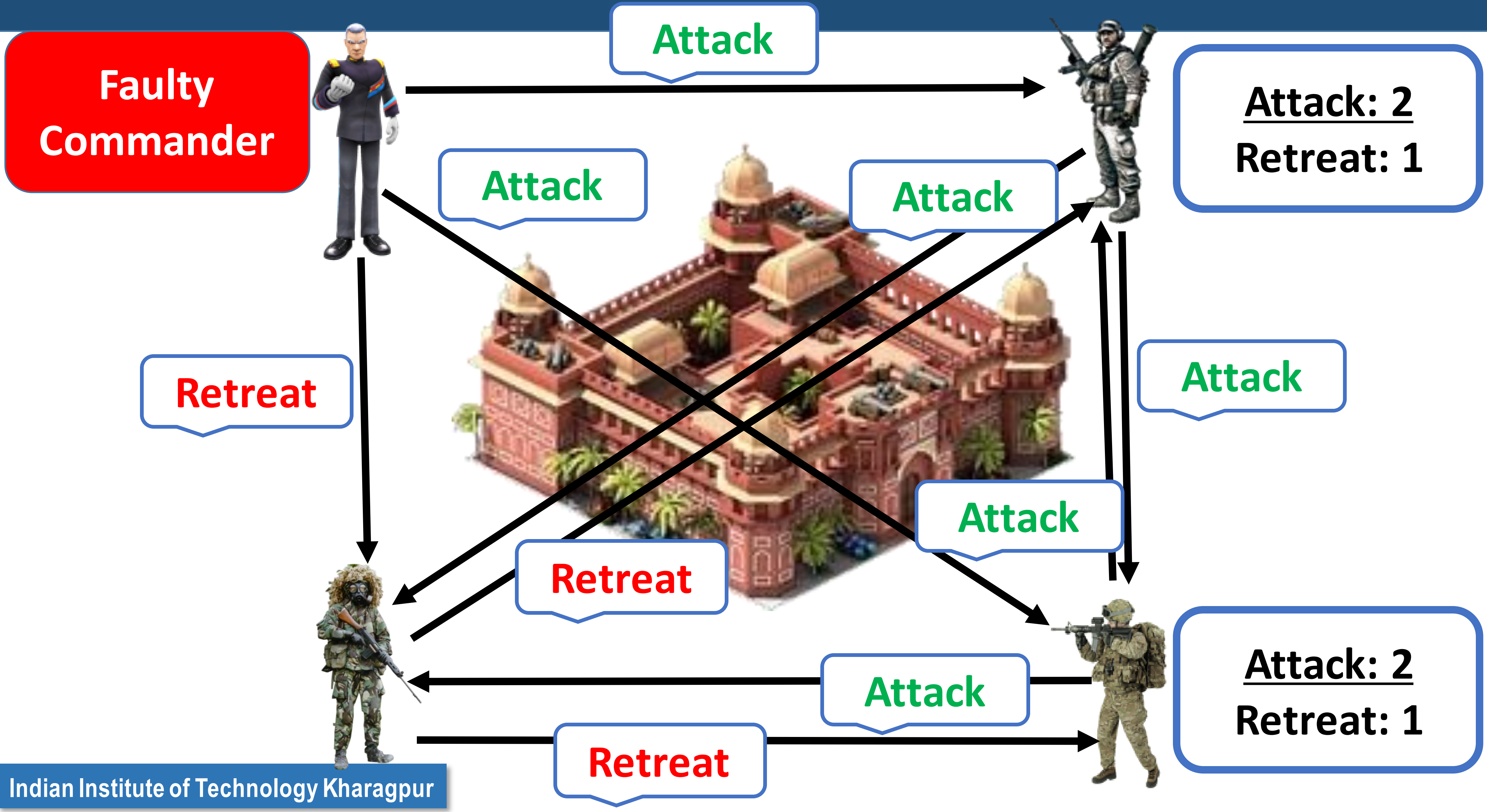
Byzantine Generals Problem – Three Lieutenants



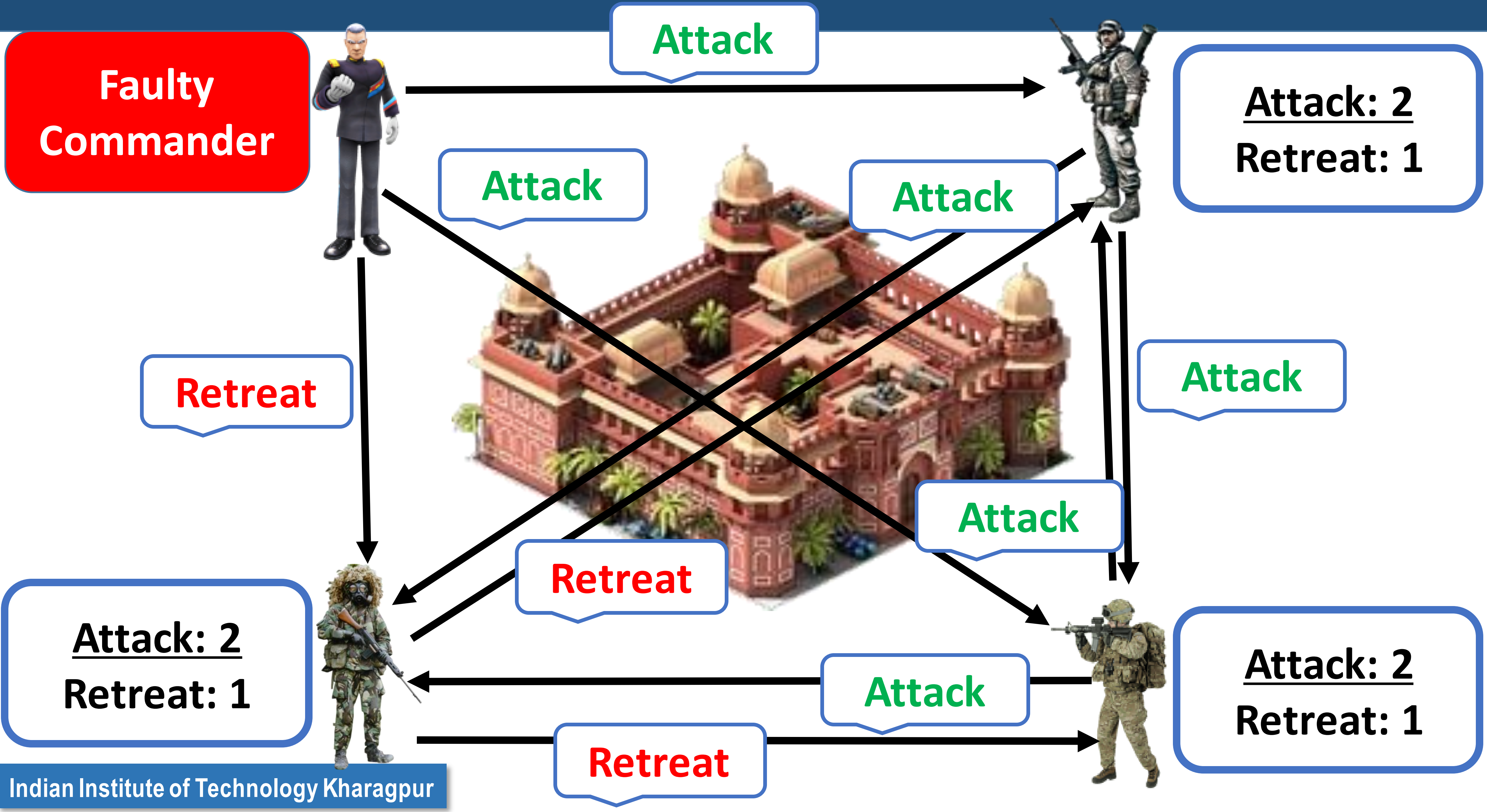
Byzantine Generals Problem – Three Lieutenants



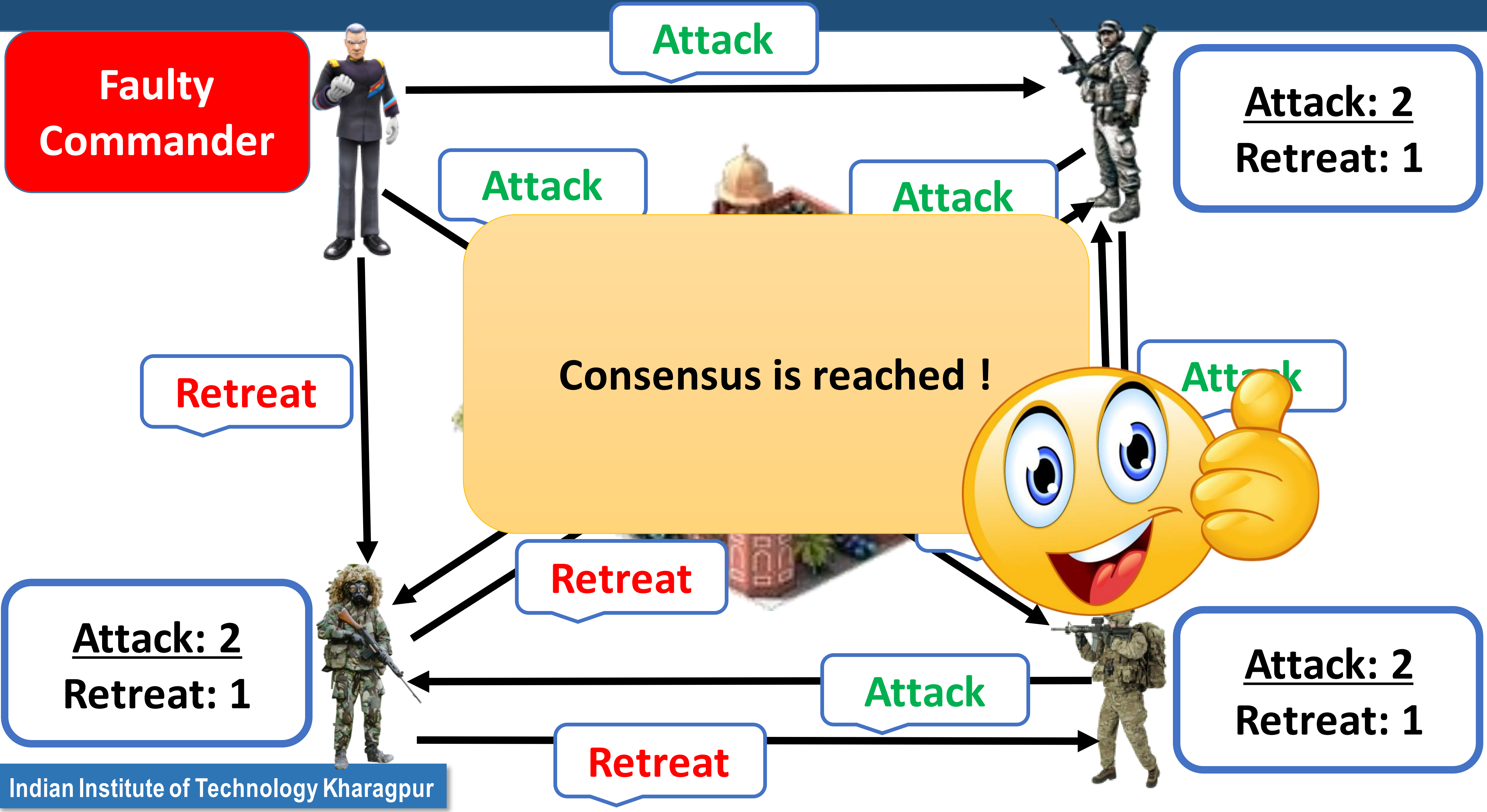
Byzantine Generals Problem – Three Lieutenants



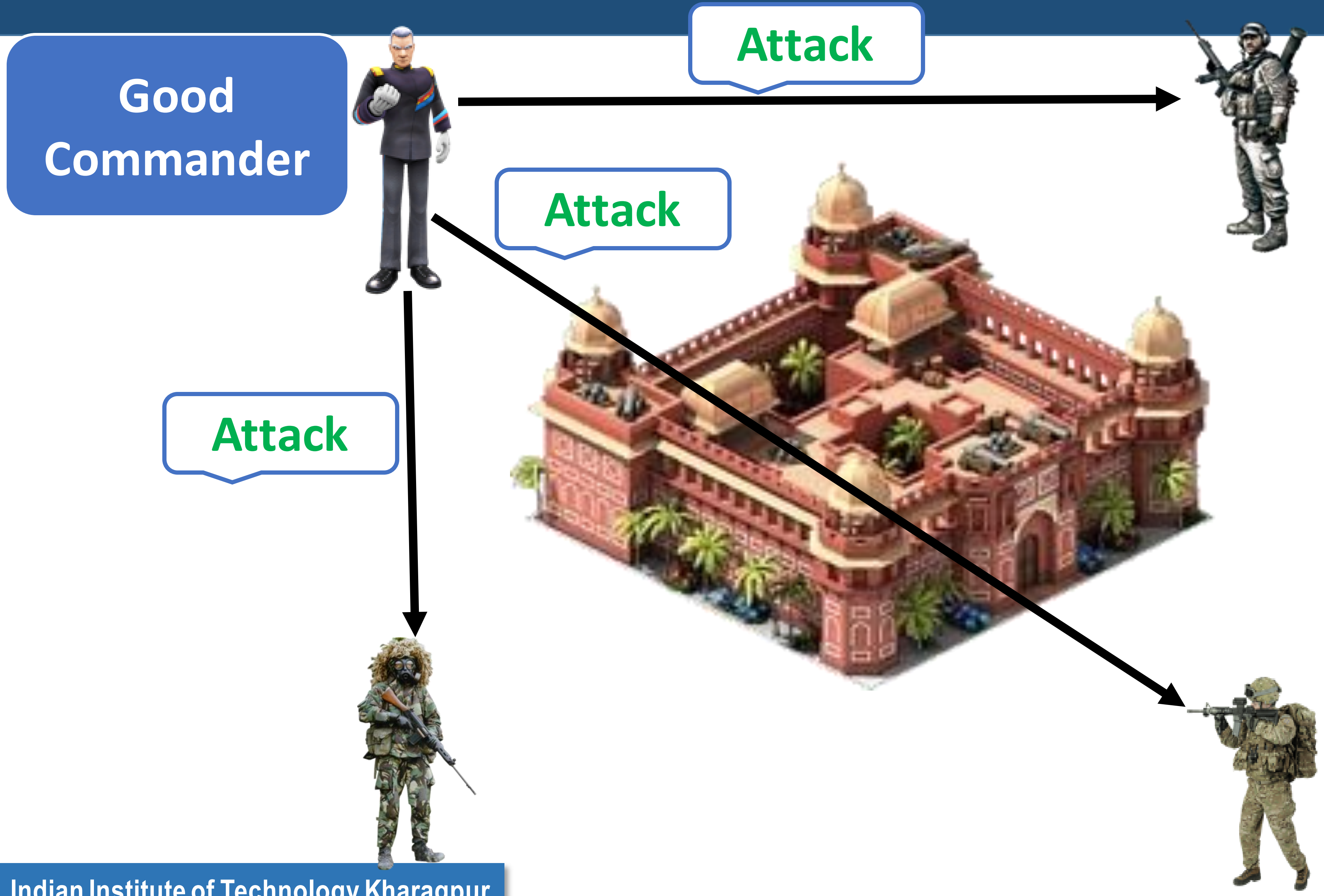
Byzantine Generals Problem – Three Lieutenants



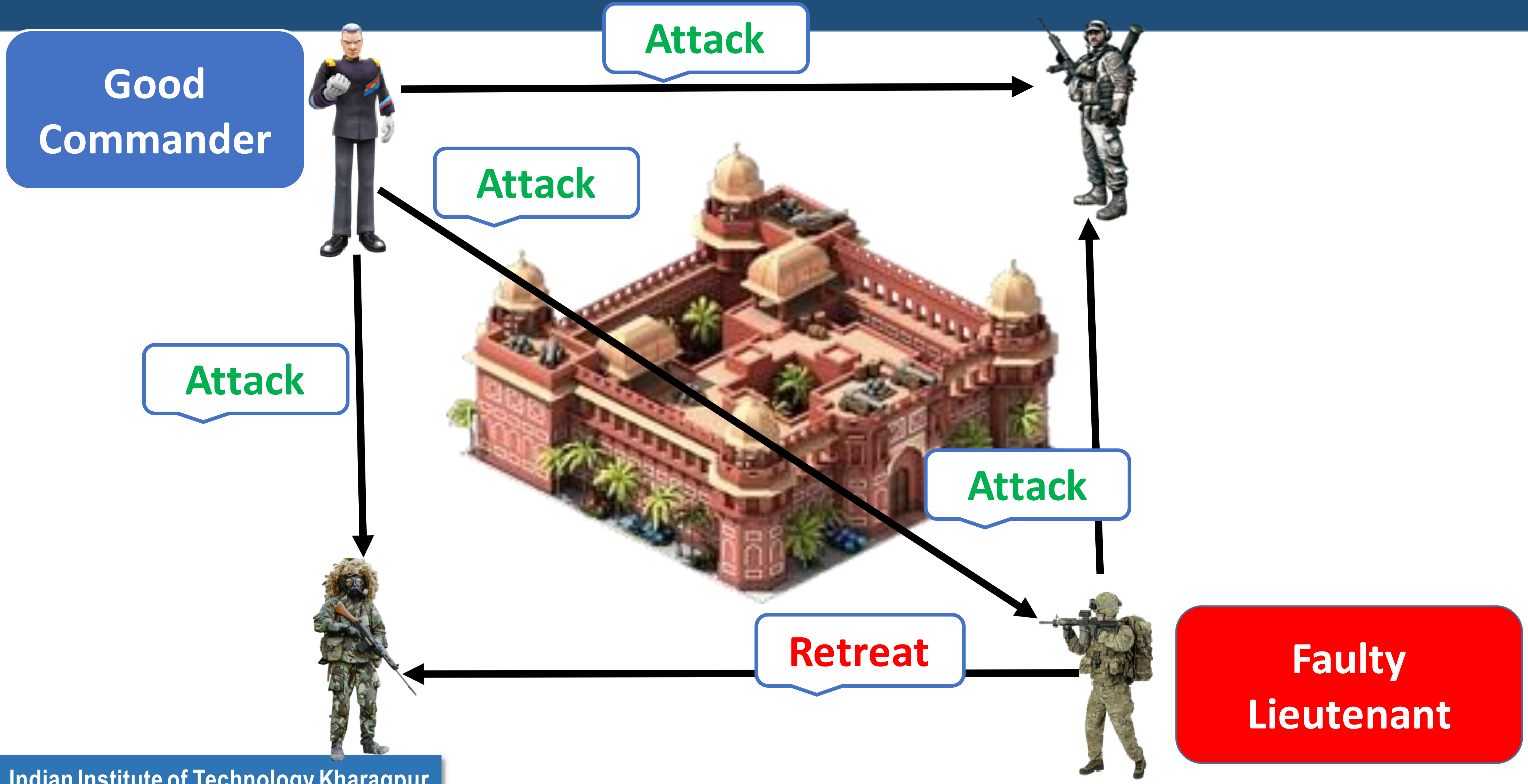
Byzantine Generals Problem – Three Lieutenants



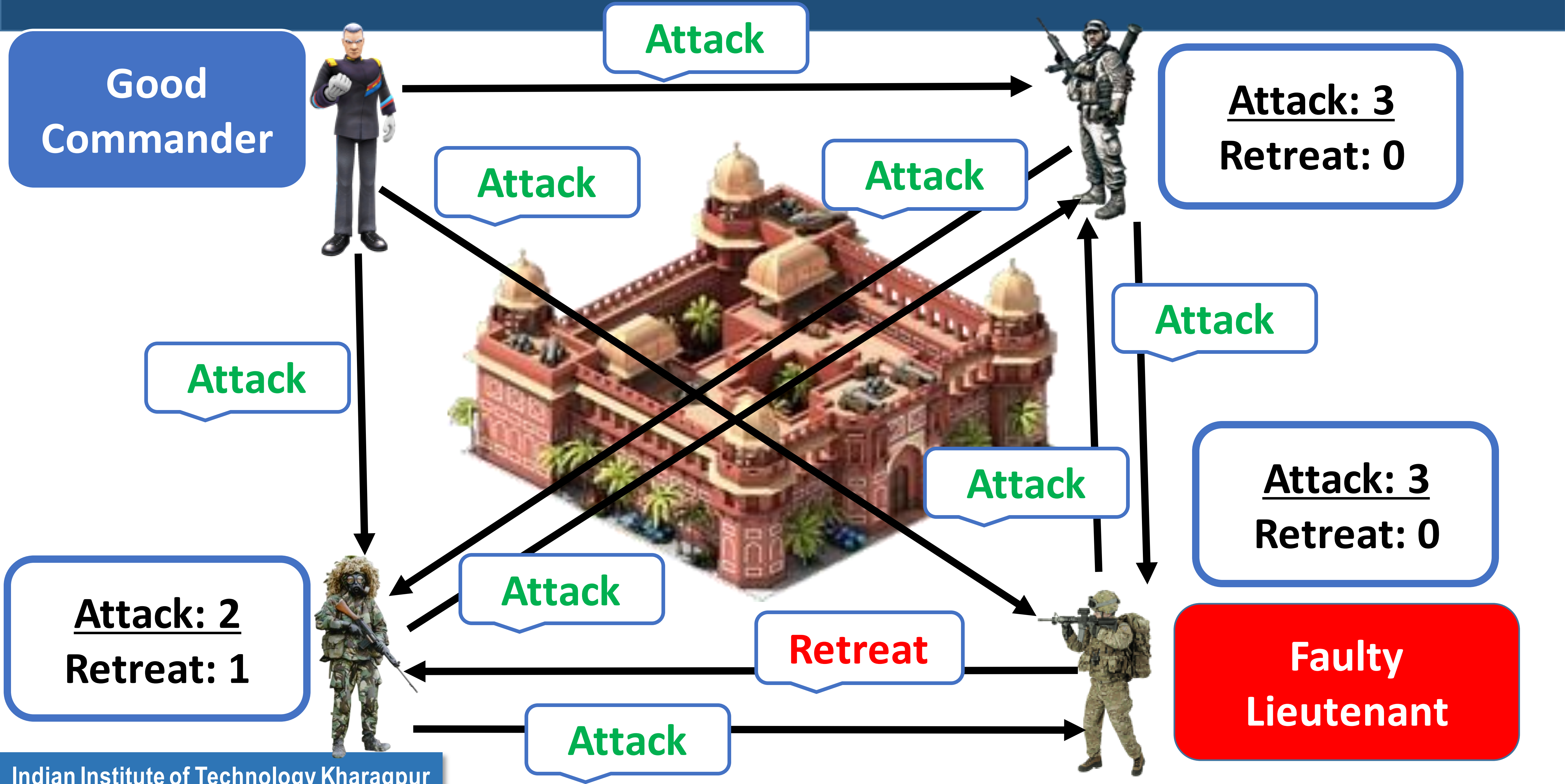
Byzantine Generals Problem – Three Lieutenants



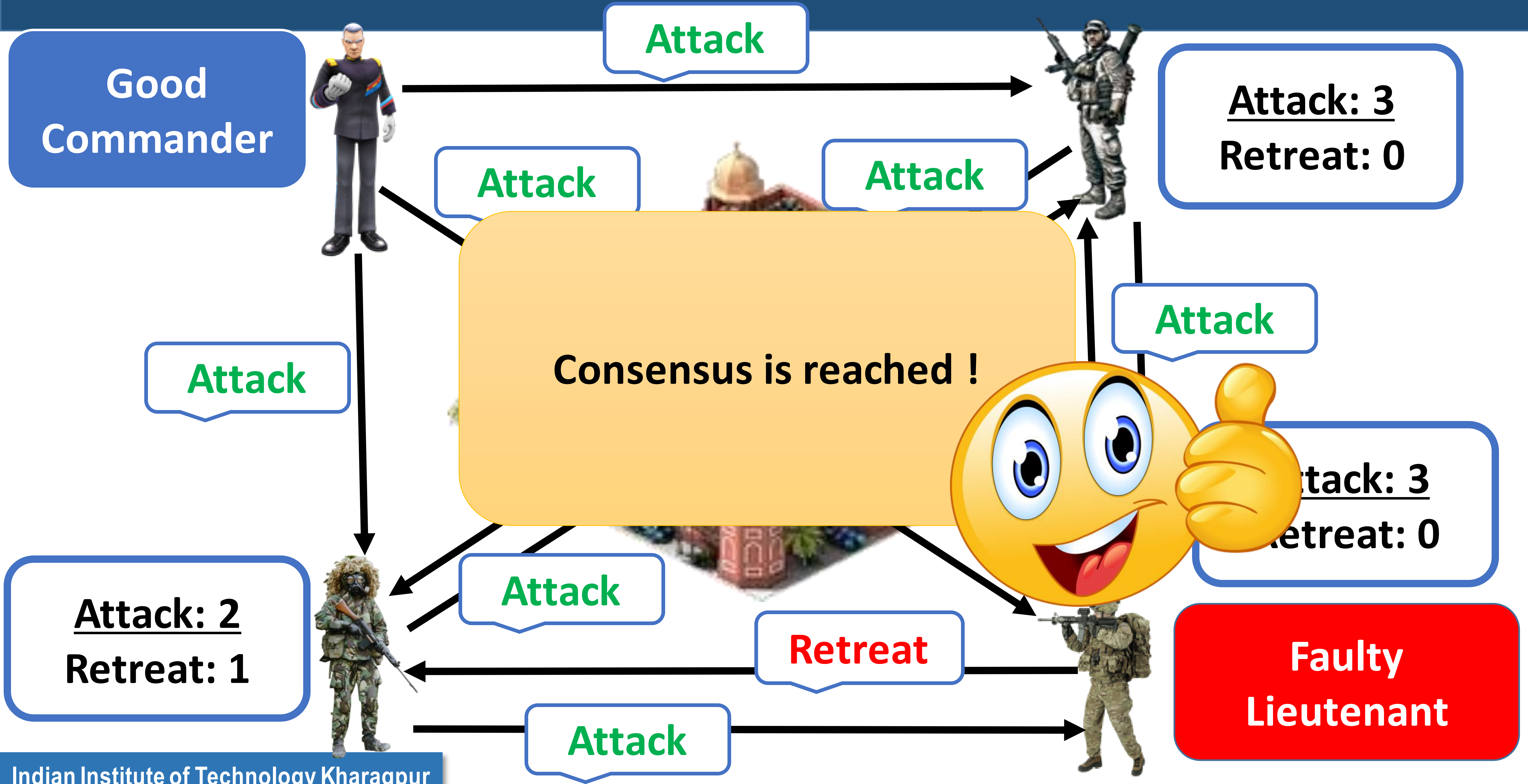
Byzantine Generals Problem – Three Lieutenants



Byzantine Generals Problem – Three Lieutenants



Byzantine Generals Problem – Three Lieutenants



Asynchronous Byzantine Agreement

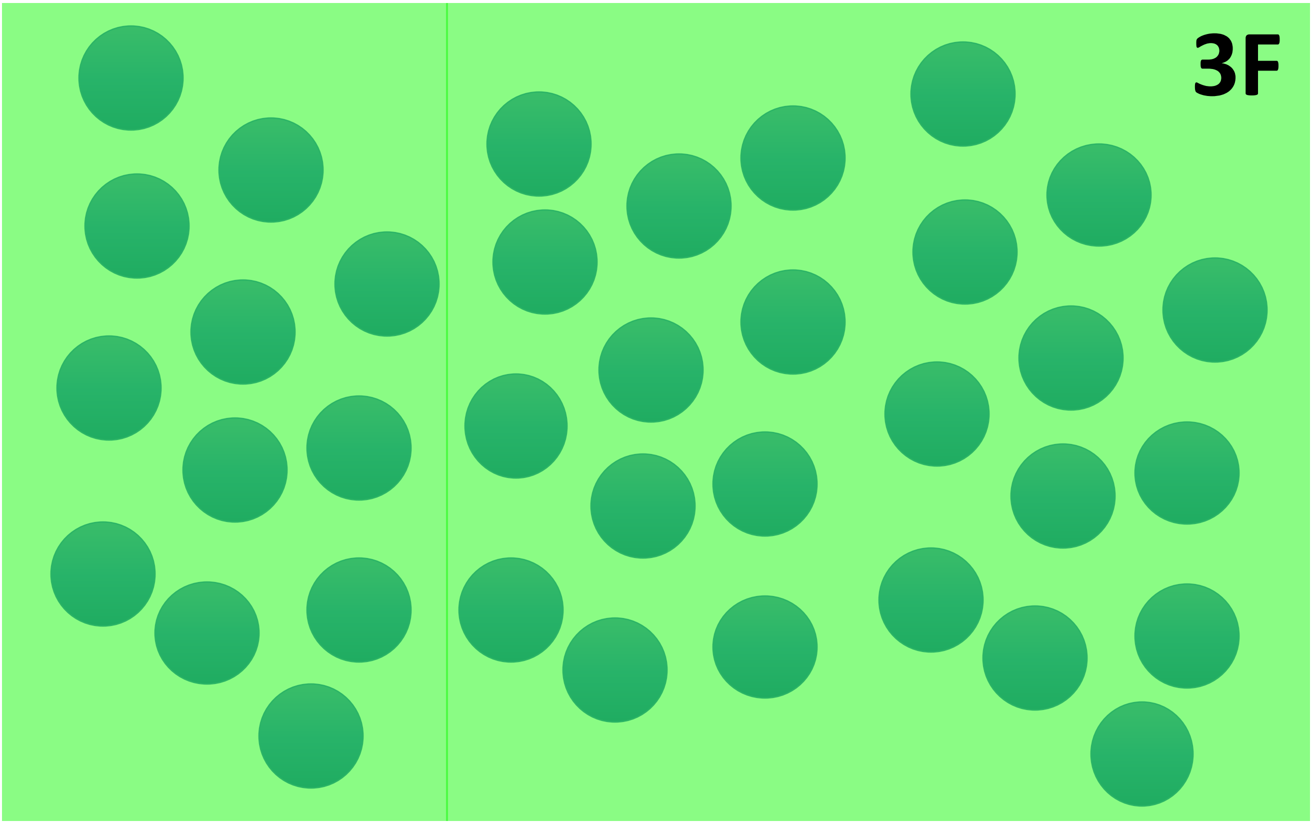
- F faulty processes – need at least 3F + 1 processes to reach consensus

Asynchronous Byzantine Agreement

- F faulty processes – need at least 3F + 1 processes to reach consensus
  - Faulty processes create partition in the network

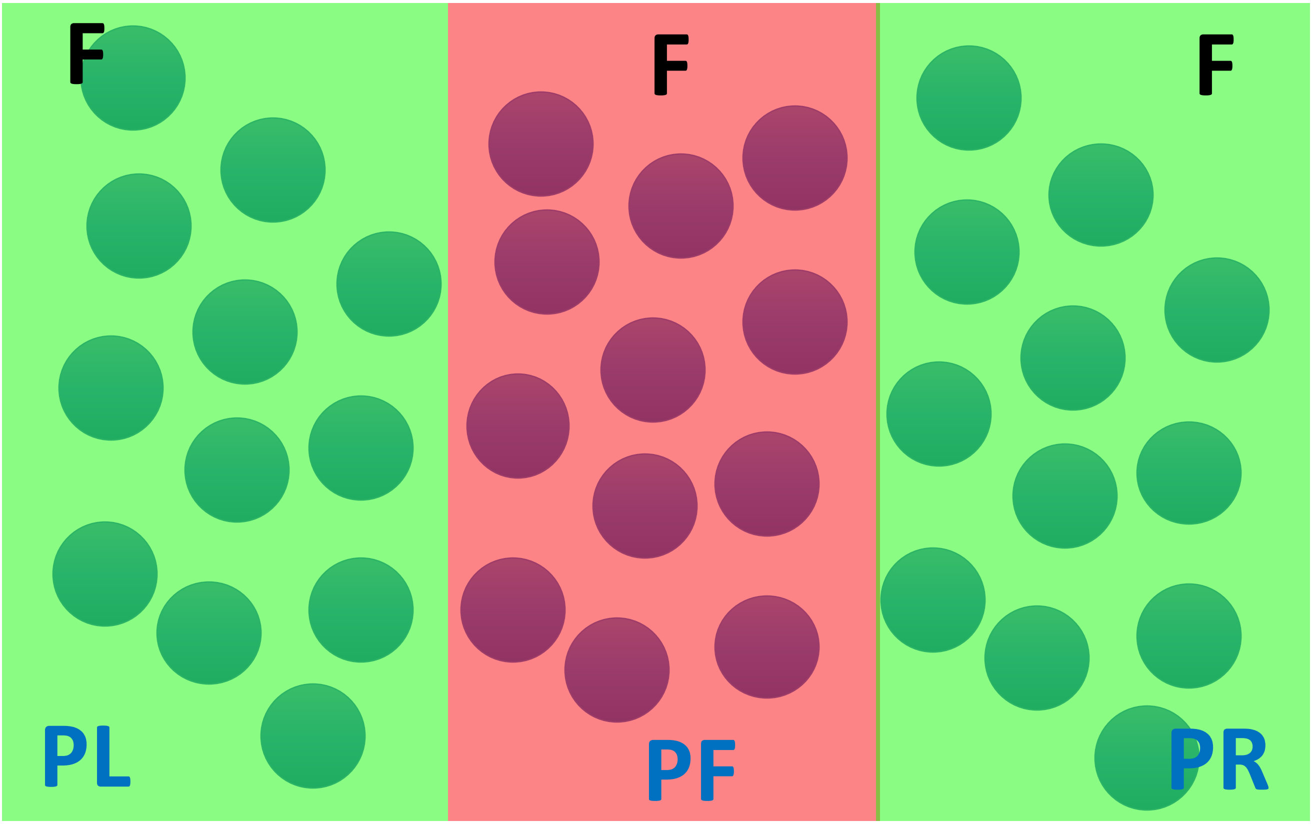
Asynchronous Byzantine Agreement

- F faulty processes – need at least 3F + 1 processes to reach consensus
  - Faulty processes create partition in the network



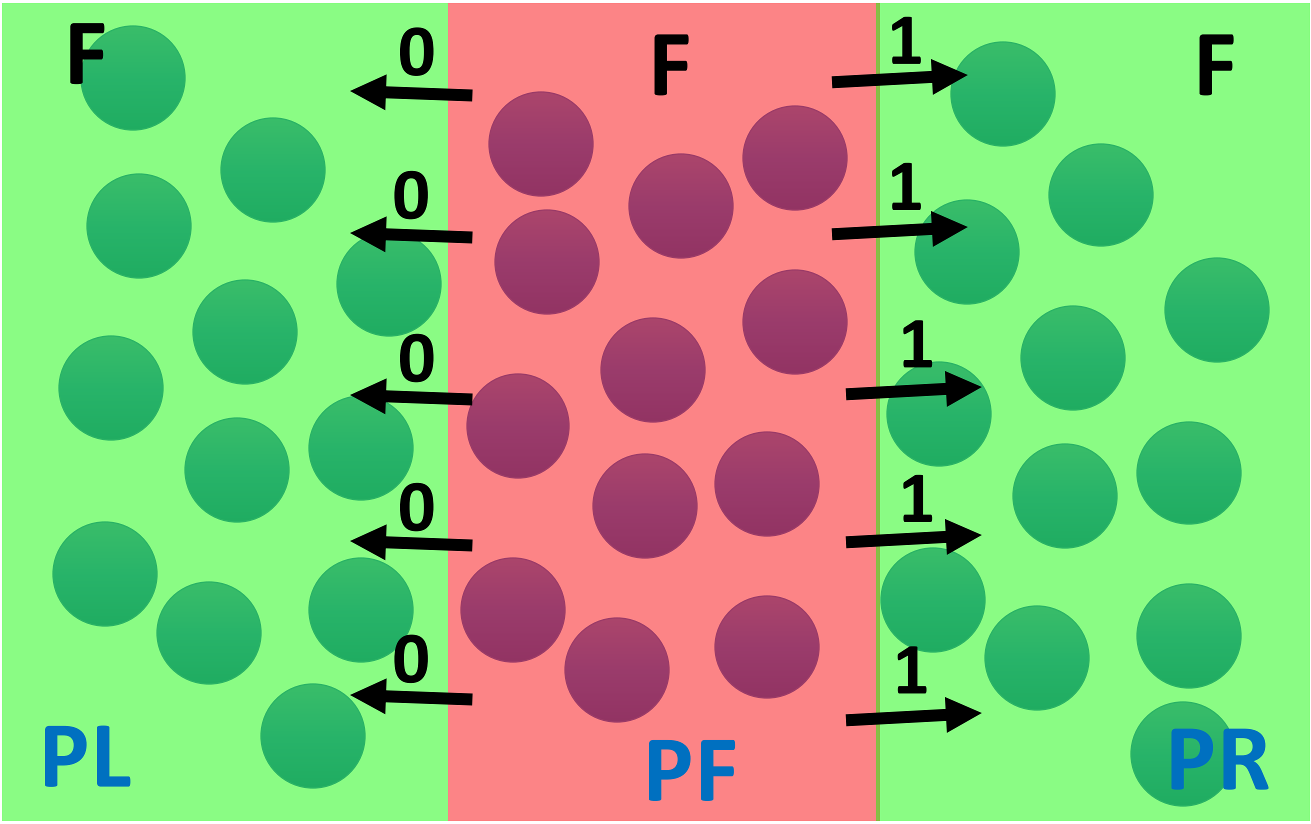
Asynchronous Byzantine Agreement

- F faulty processes – need at least 3F + 1 processes to reach consensus
  - Faulty processes create partition in the network



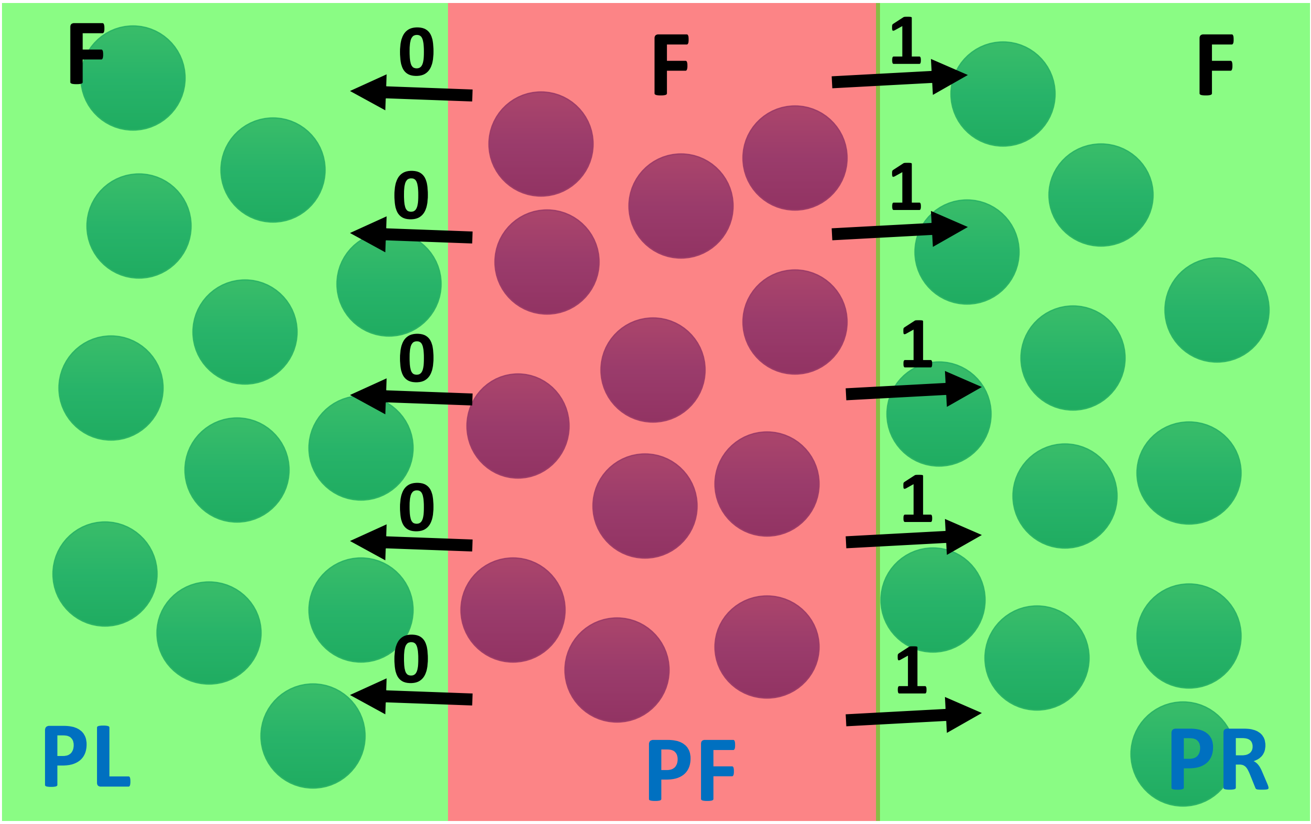
Asynchronous Byzantine Agreement

- F faulty processes – need at least 3F + 1 processes to reach consensus
  - Faulty processes create partition in the network



Asynchronous Byzantine Agreement

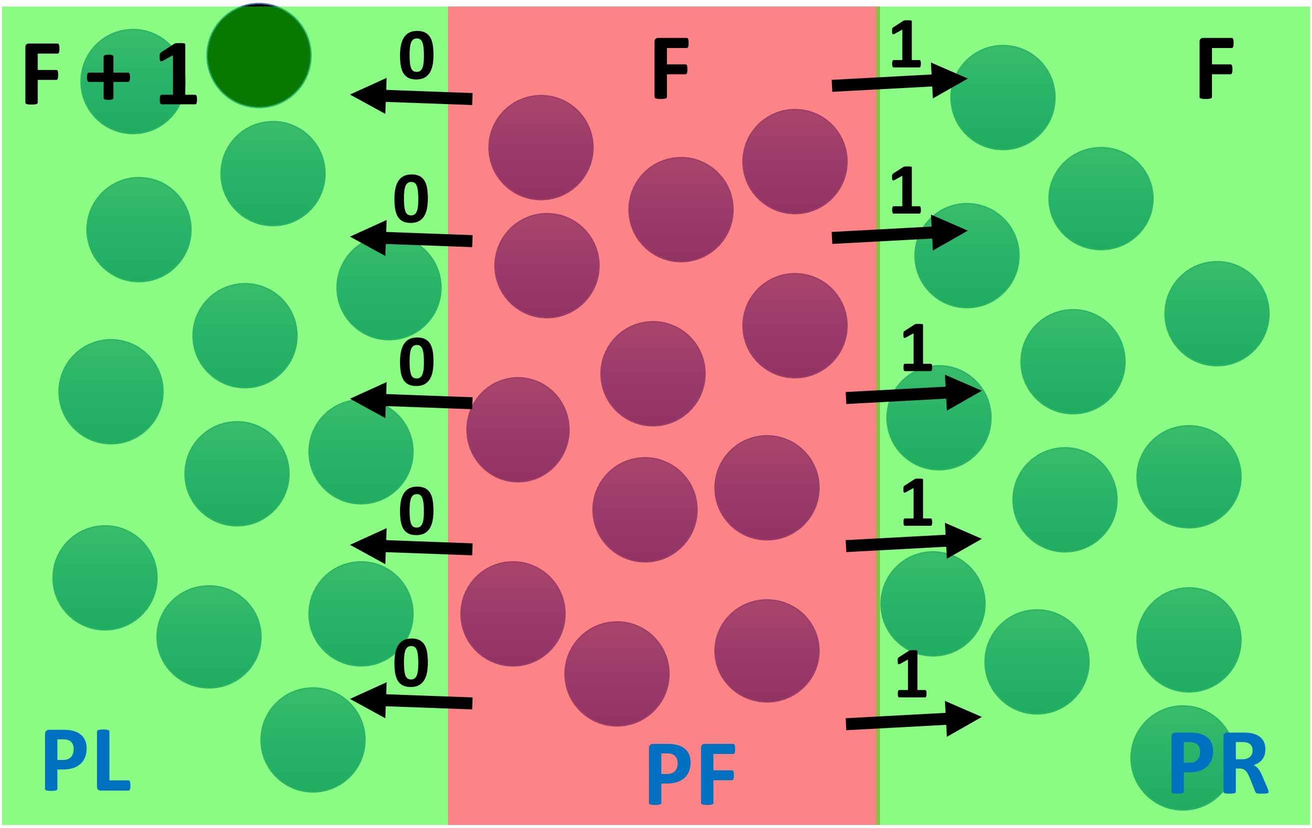
- F faulty processes – need at least 3F + 1 processes to reach consensus
  - Faulty processes create partition in the network



Either PL or PR must break the tie

Asynchronous Byzantine Agreement

- F faulty processes – need at least 3F + 1 processes to reach consensus
  - Faulty processes create partition in the network



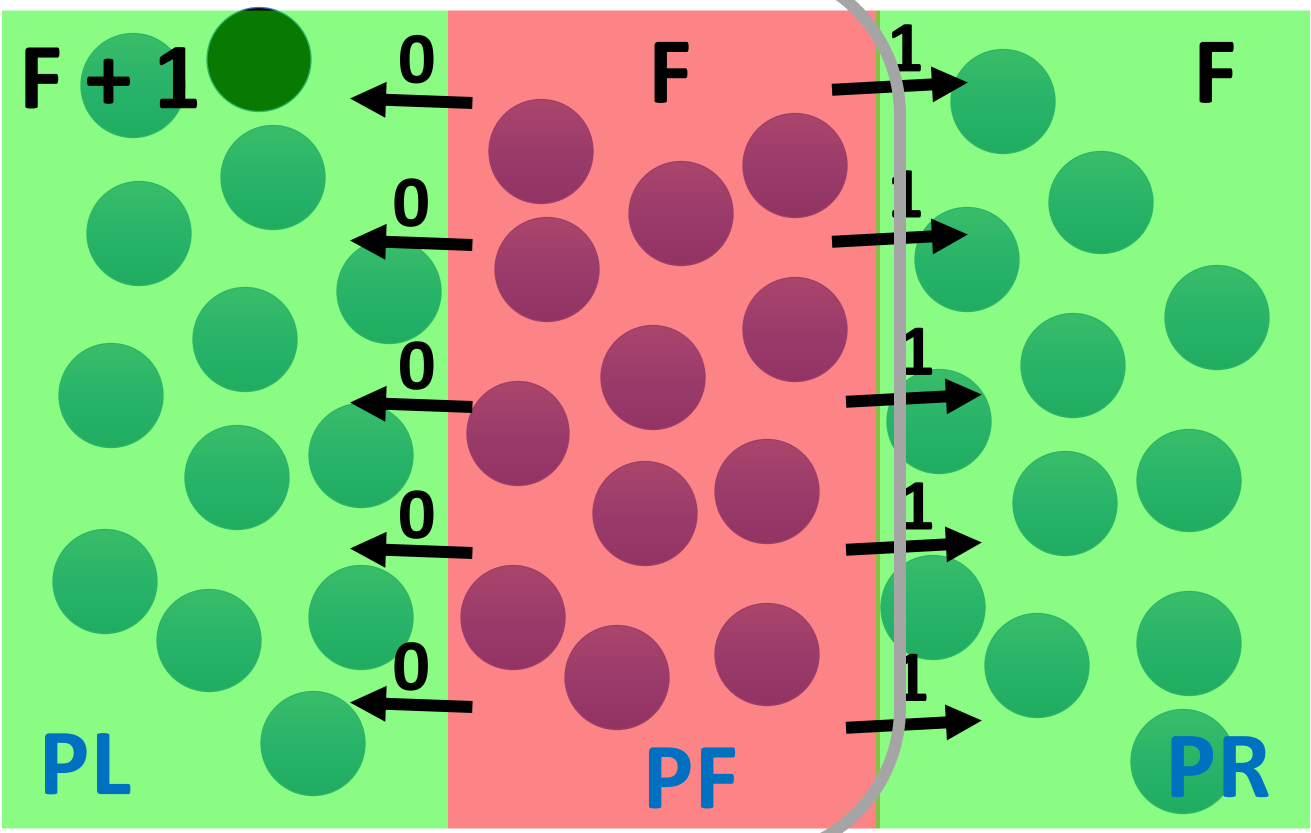
Put one additional node to PL / PR

Asynchronous Byzantine Agreement

- F faulty processes – need at least 3F + 1 processes to reach consensus
  - Faulty processes create partition in the network

Put one additional node to PL / PR

Breaks the tie to reach consensus



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.

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 a large number of distributed applications including blockchain
  - Tendermint
  - Hyperledger Fabric

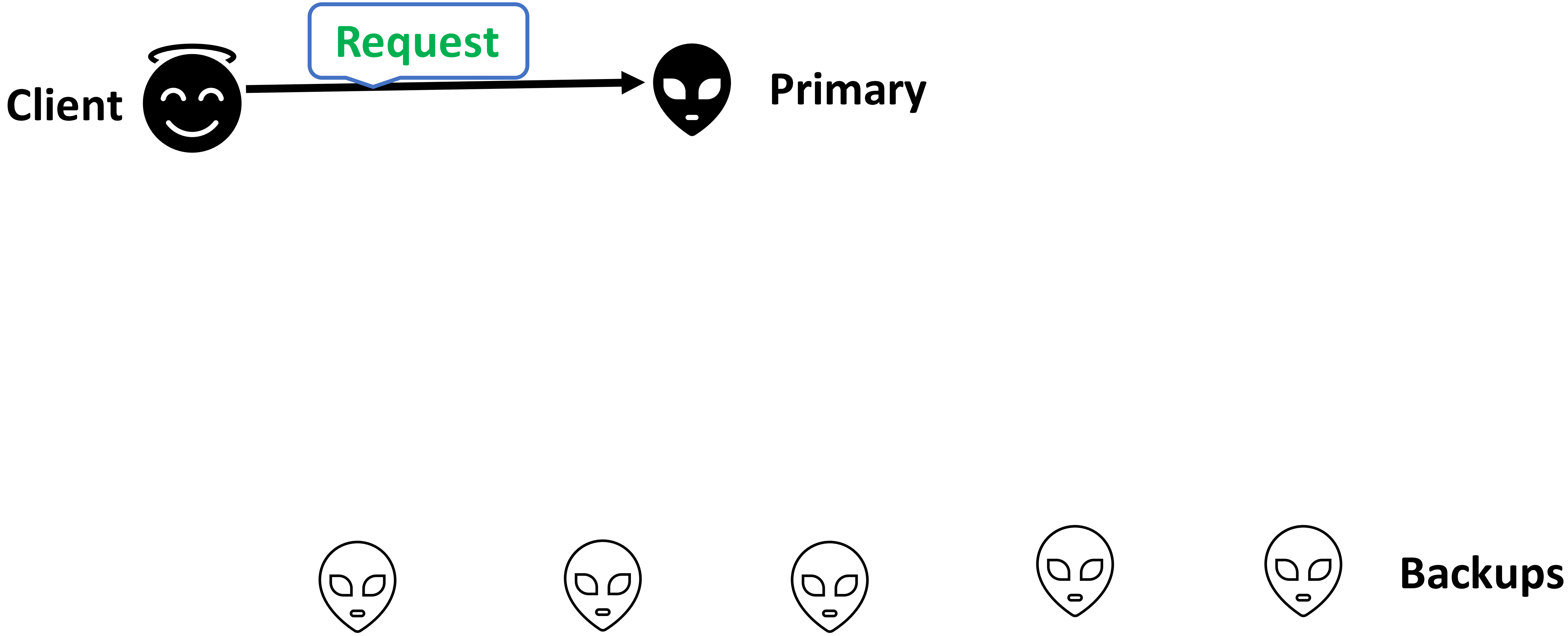
- Uses cryptographic techniques to make the messages tamper-proof

PBFT Overview

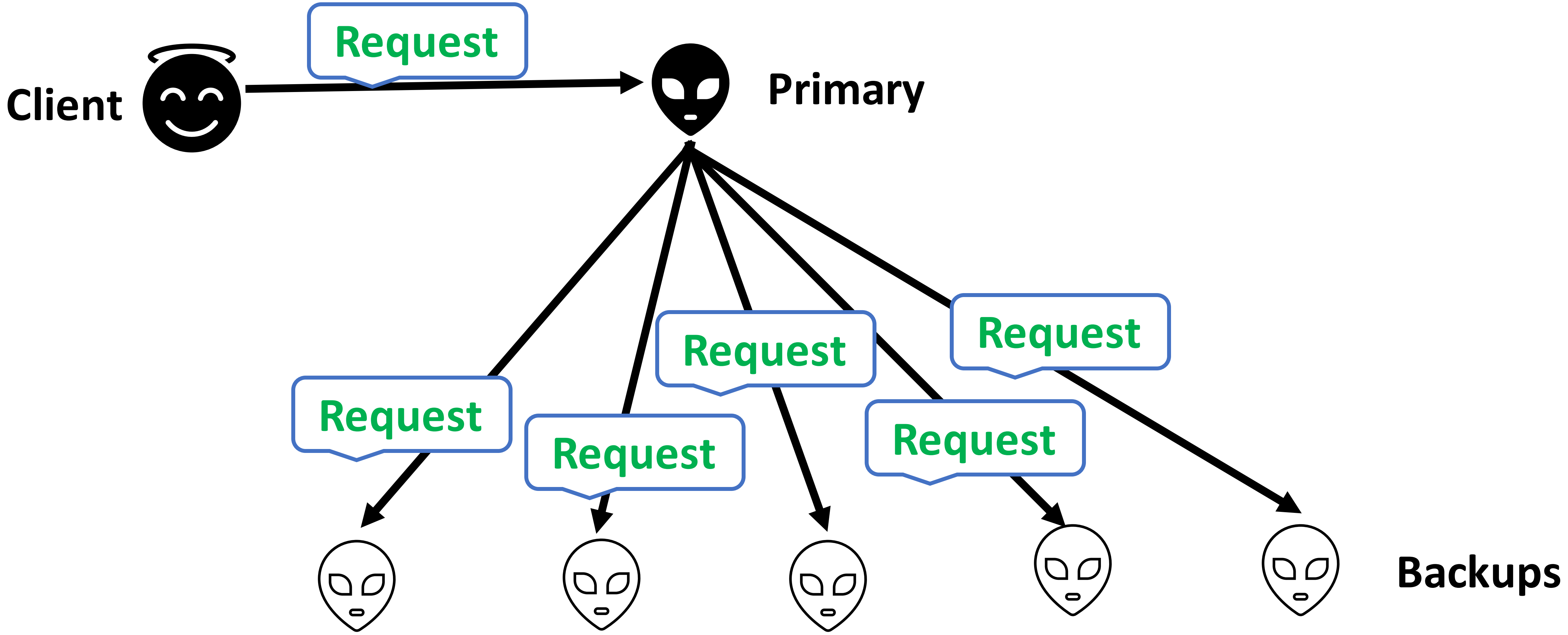
- Based on State Machine Replication
  - Considers 3F + 1 replicas where F can be the maximum number of faulty replicas

- 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

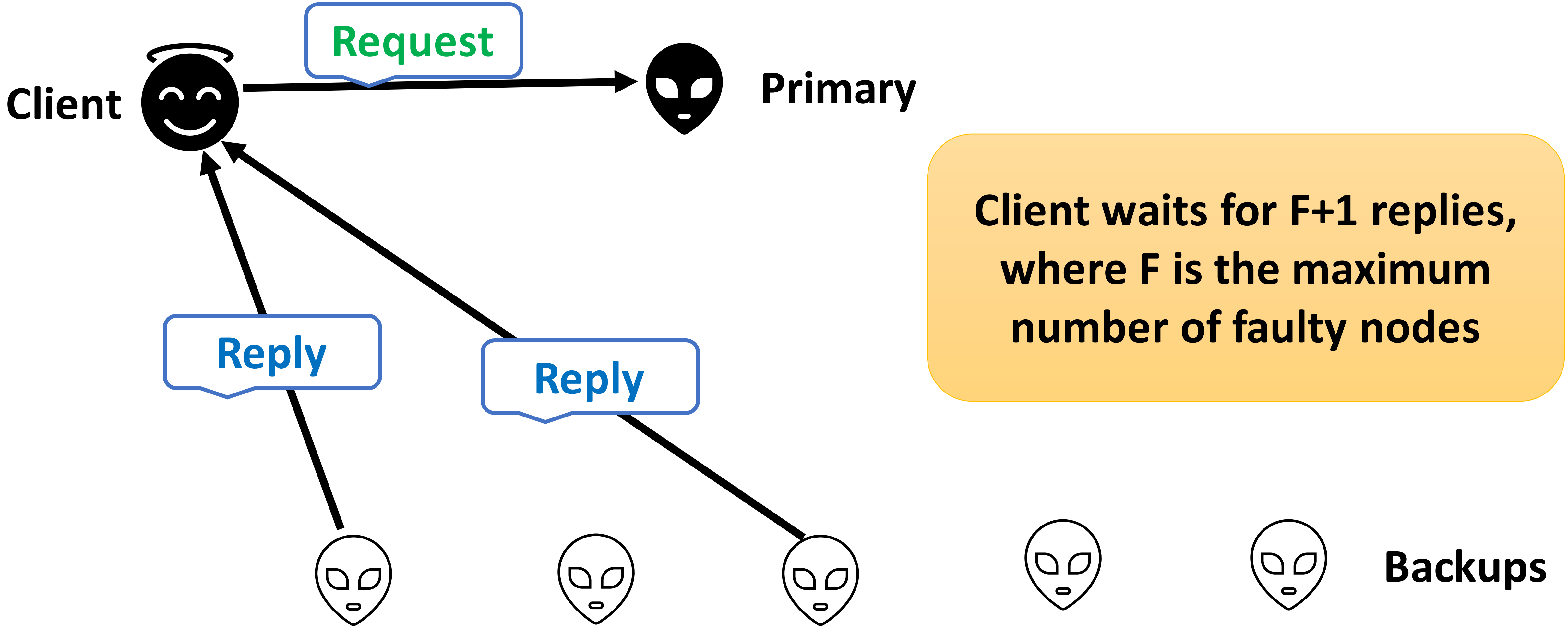
PBFT – Broad Idea



PBFT – Broad Idea

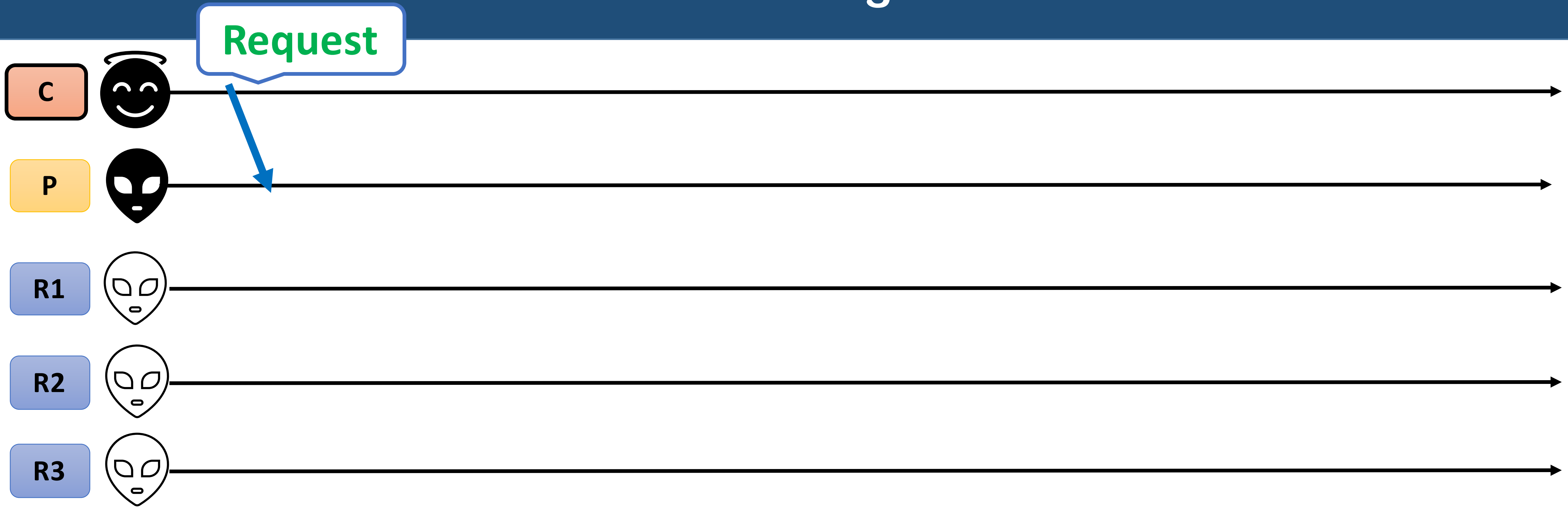


PBFT – Broad Idea



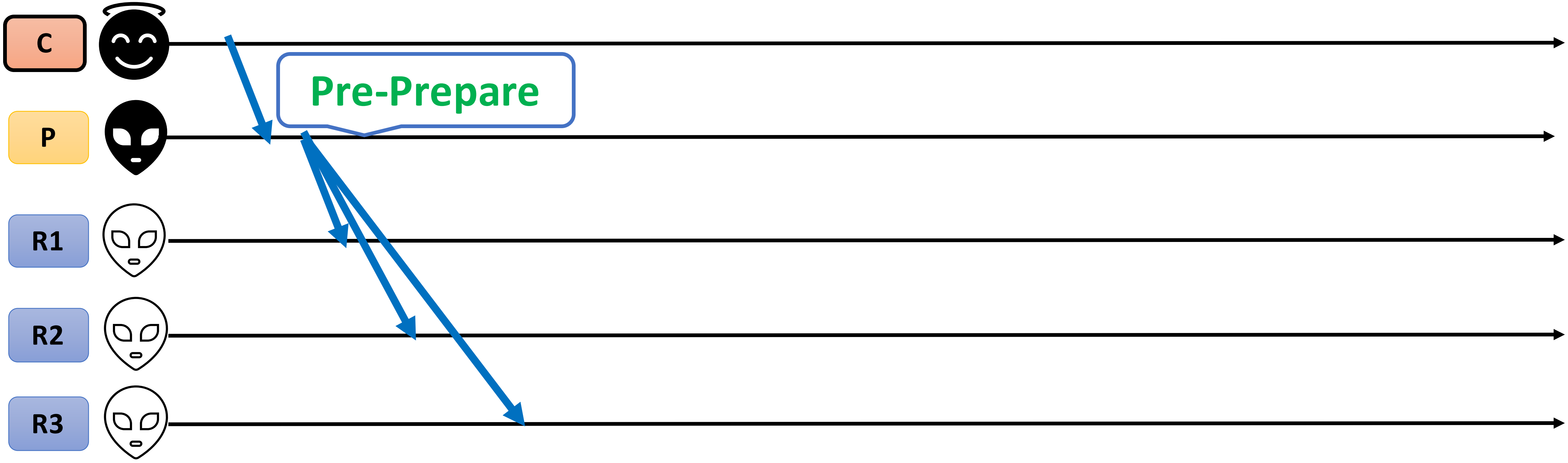
Client waits for F+1 replies, where F is the maximum number of faulty nodes

PBFT – The Algorithm



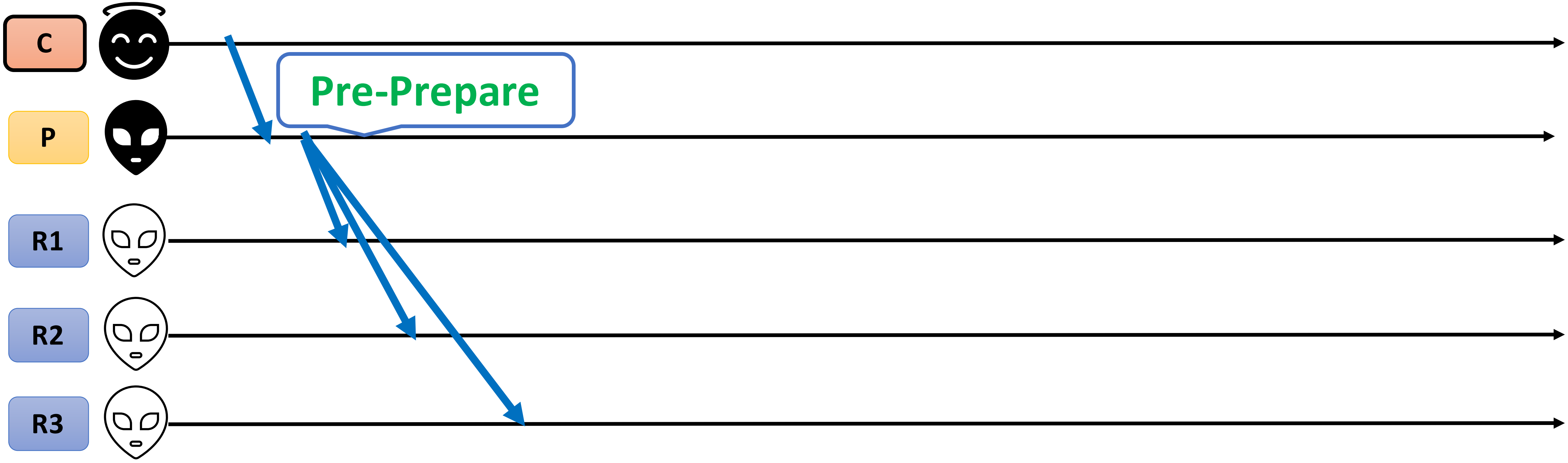
- The protocol starts by the client sending a Request message to the primary
- The primary collects all the Request messages from different clients and order them based on certain pre-defined logic

PBFT – The Algorithm



- Primary assigns a sequence number *n* to the Request (or a set of Requests) and multicast a message **<<PRE-PREPARE, *v*, *n*, *d*>$_{\beta\_p}$, *m*>** to all the backups
  - *v* is the current view number, *d* is the message digest, m is the message
  - β_p is the private key of the primary

PBFT – The Algorithm



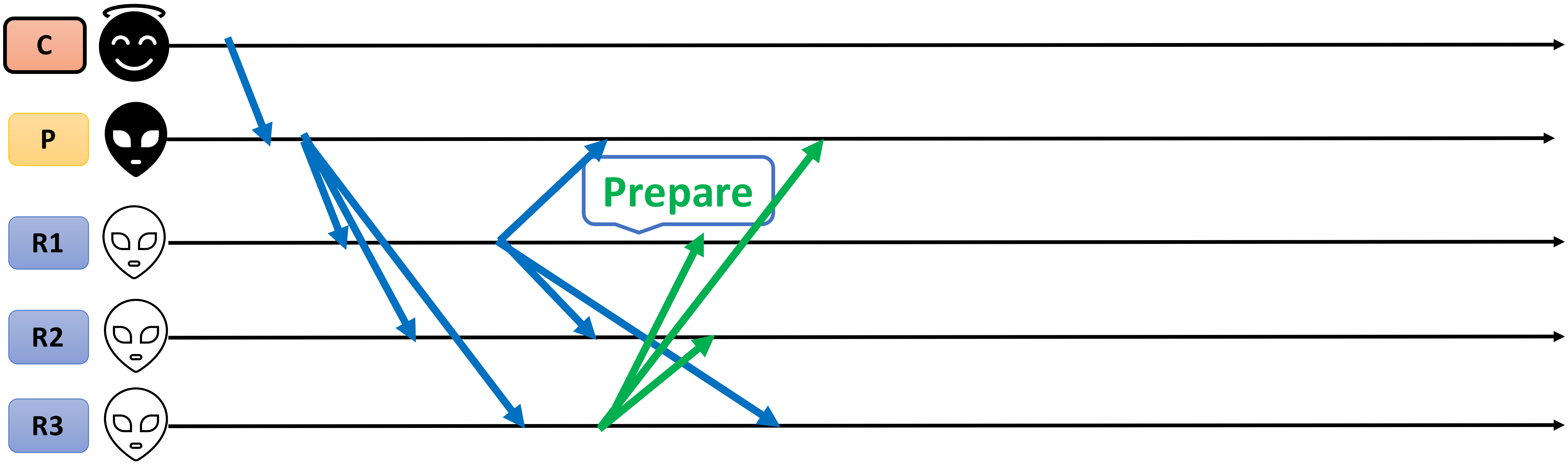
- Pre-prepare works as a proof that the Request was assigned a sequence number *n* for the view *v*

PBFT – The Algorithm

A backup accepts the Pre-prepare message, if

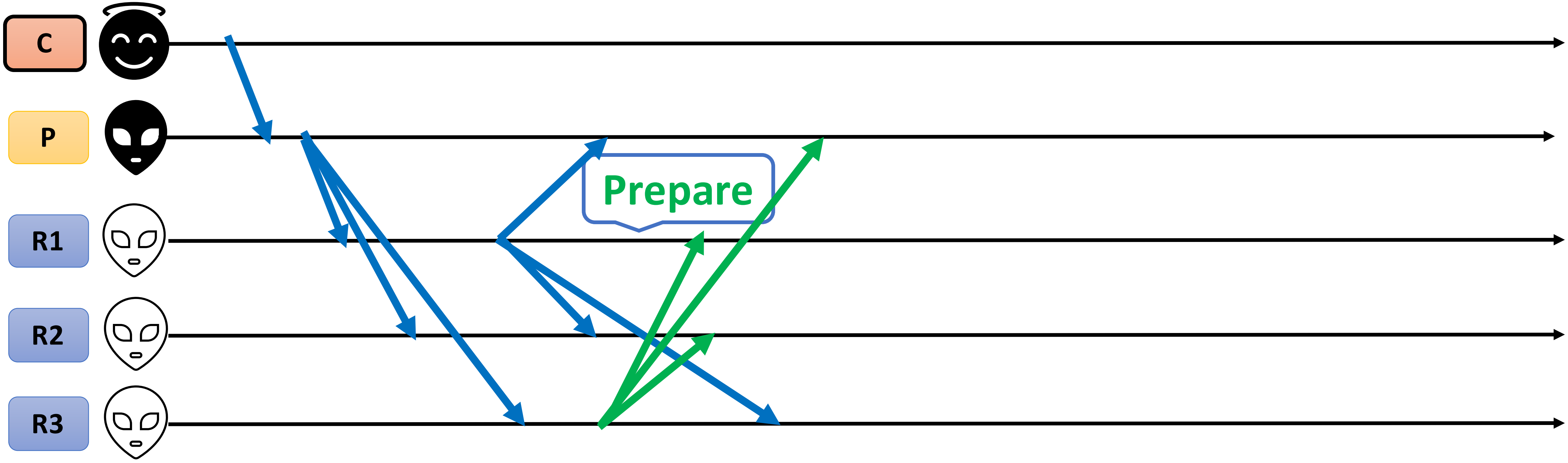
- The signature is correct and *d* is the digest of the message *m*
- The backup is in view *v*
- It has not received a different Pre-Prepare message with sequence *n* and view *v* with a different message digest
- The sequence number is within a threshold (the message is not too old – prevents a reply 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* : <PREPARE, *v*, *n*, *d*, *k*>$_{\beta\_k}$

PBFT – The Algorithm

Prinary 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 Dissimination 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 atleast ***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 atleast ***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 atleast $3f + 1$ 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

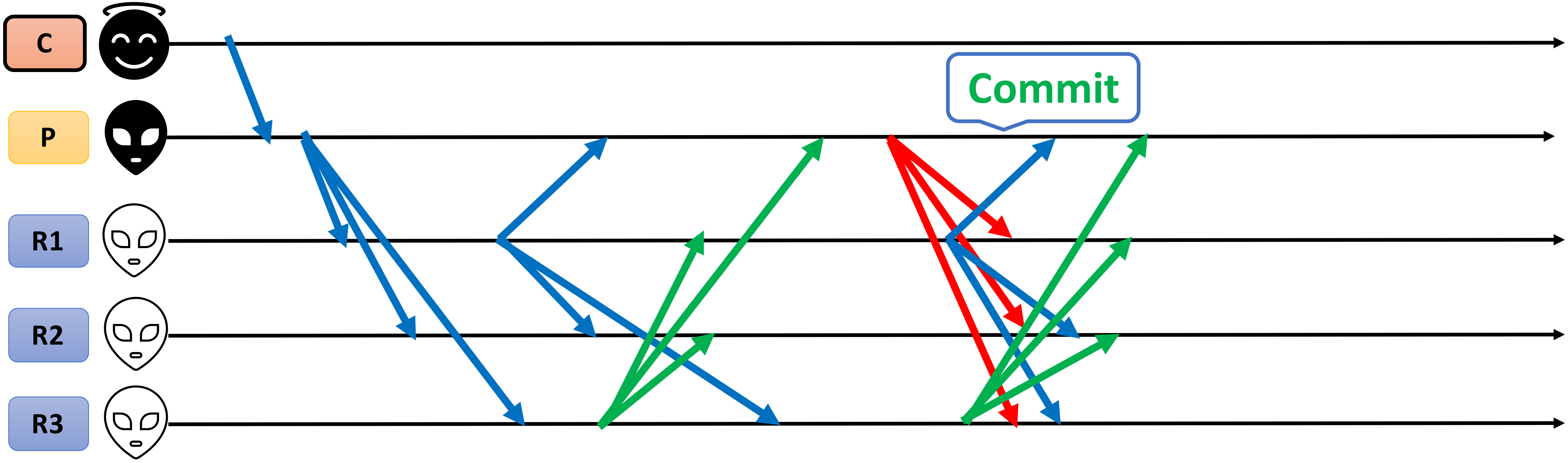
Quorum in PBFT

- You have *f* number of faulty nodes – you need atleast ***3f + 1*** replicas to reach consensus
  - But yo ts, or Byz
- **Case 2: A**
  - You m om Byzan
  - Suffici you still have ***f***
  - You cannot wait for ***f+1***, the first ***f*** might be all faulty

Remember, you are on an asynchronous channel – messages get delayed and can be received out of order

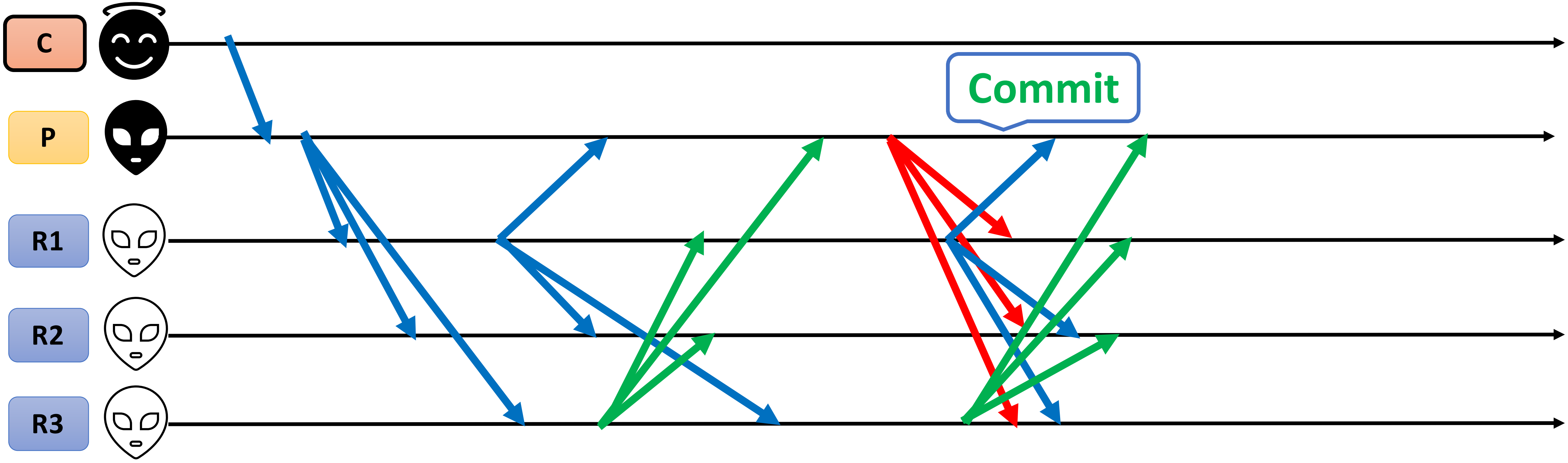
Wait untill you receive *2f + 1* Prepare messages – once you received *2f + 1* votes, you can safely take a decision based on majority voting

PBFT – The Algorithm



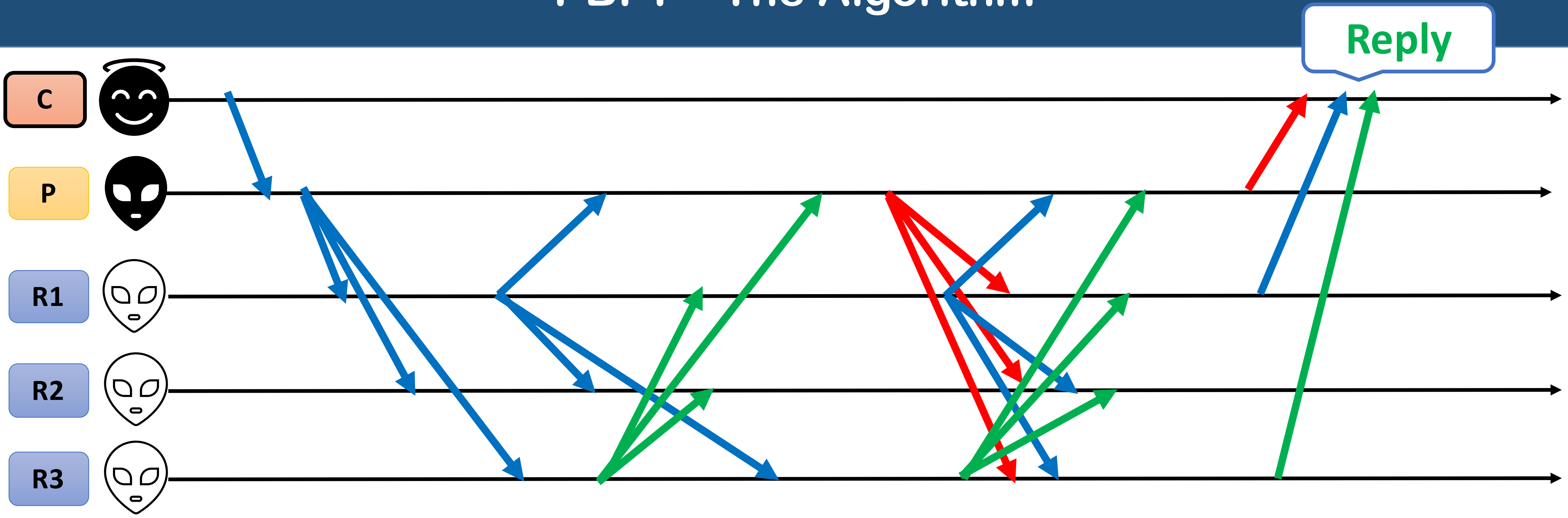
- Message format for replica *k* : <COMMIT, *v*, *n*, *d*, *k*>$_{\beta\_k}$

PBFT – The Algorithm



- Message format for replica k : <COMMIT, *v*, *n*, *d*, *k*>$_{\beta\_k}$
- 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* : **<COMMIT, *v*, *n*, *d*, *k*>**$_{\beta\_k}$
- 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

- Unline 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

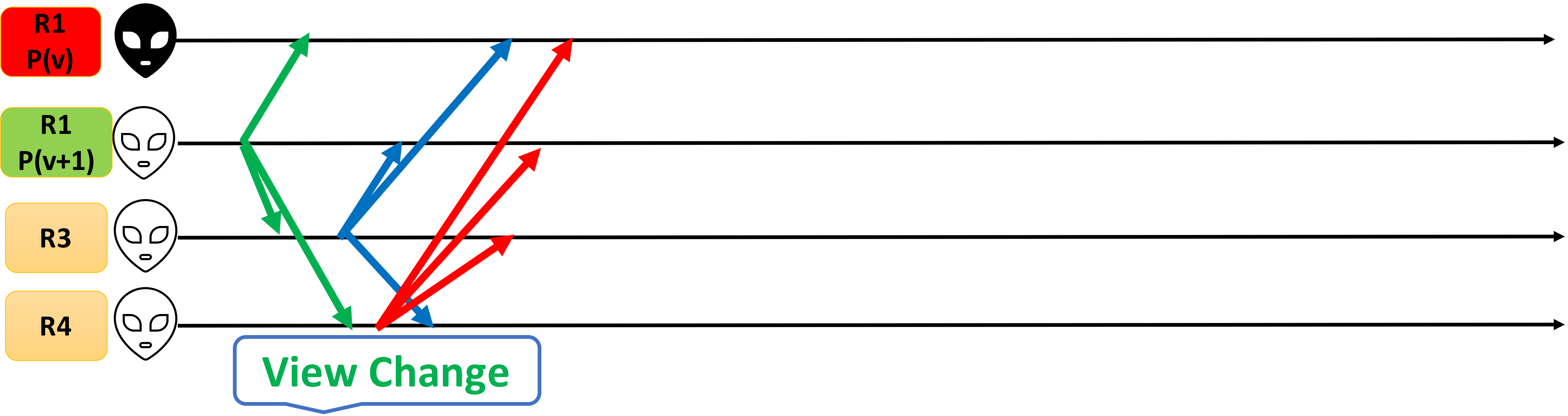
- View changes are triggered by timeouts (weak synchrony assumption)
  - Prevent backups from waiting indefinitely for requests to execute

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

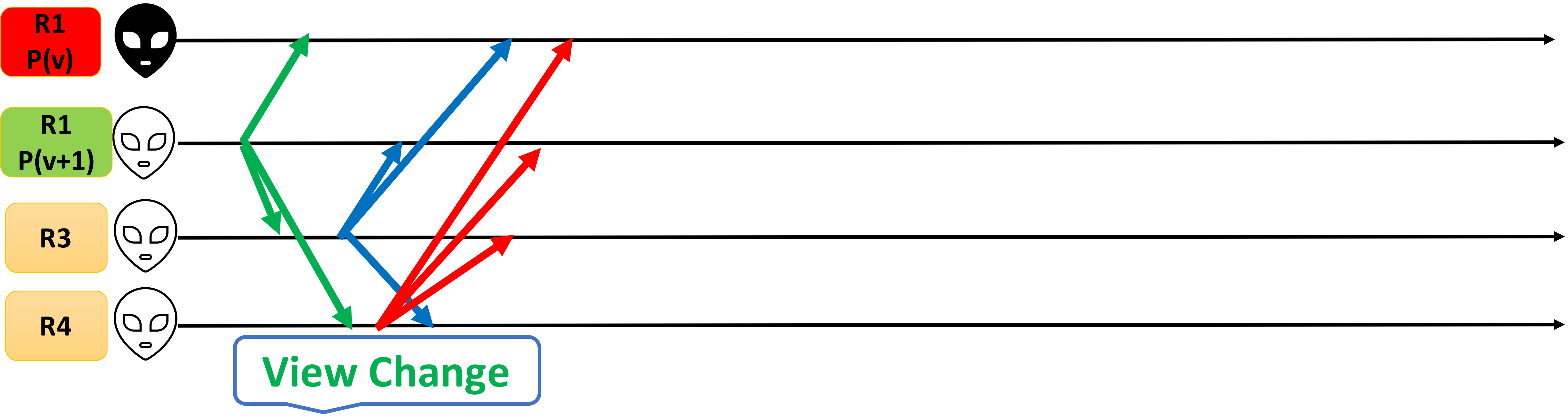
- If the timer expires at view v, backup starts a **View Change** to move to the view $v + 1$

The View Change Protocol



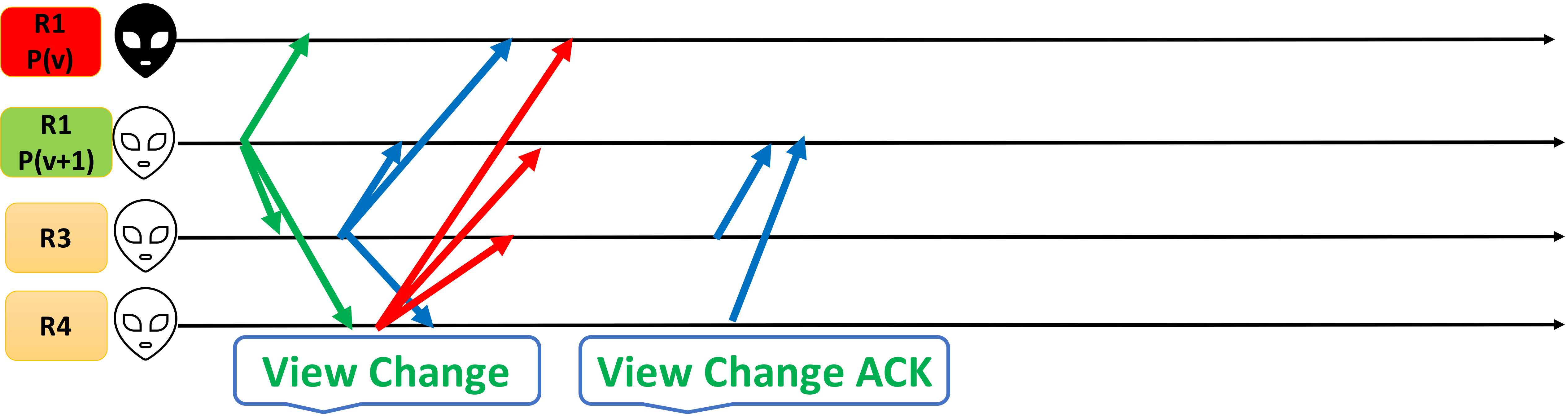
- Multicast the view change message **<VIEW-CHANGE, *v+1, n, C, P, k*>$_{\beta\_k}$**
  - *n* is the sequence number of last stable checkpoint *s* known to *k*
  - *C* is a set of *2f + 1* valid checkpoint messages corresponding to *s*
  - *P* is a set containing a set P_m for each request *m* that prepared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

The View Change Protocol



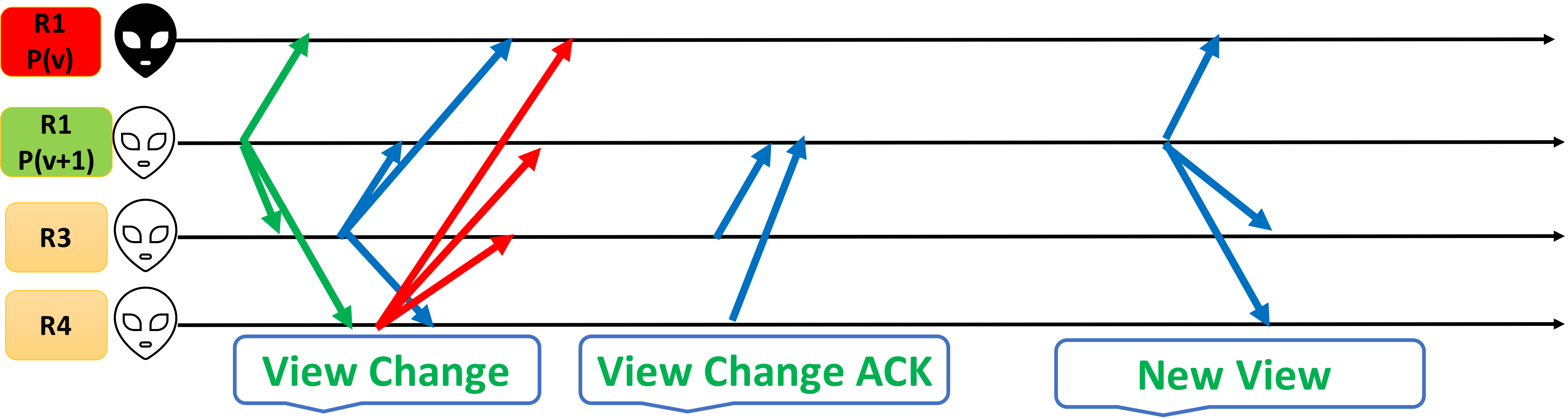
- The new view is initiated after receiving *2f + 1* View Change messages
- Next primary selection
  - Round Robin (Hyperledger Sawtooth)
  - Leader election (Hyperledger Fabric)

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

thank you!