Dependable and Distributed Systems

Professor Matthew Leeke
School of Computer Science
University of Birmingham

Topic 7 - Byzantine Generals Problem

Roadmap

We have studies the distributed consensus problem

- Link failures
- Stop failures

It turns out that solving the distributed consensus problem under a Byzantine failure models gets interesting quite quickly

The Byzantine General Problem

Motivation

We studied consensus in the presence of stop failures

It is possible for nodes to failure in more malicious ways

- Important to note that malicious, in this context, does not strictly imply malice on the part of some third party
- Possible for arbitrary and malicious behaviour to be indistinguishable

Can we still achieve consensus? Are there any caveats?

The Godfather

“I have long felt that, because it was posed as a cute problem about philosophers seated around a table, Dijkstra's dining philosopher’s problem received much more attention than it deserves.”

“The popularity of the dining philosophers problem taught me that the best way to attract attention to a problem is to present it in terms of a story”

“There is a problem in distributed computing, sometimes called the Chinese Generals Problem, where two generals must agree on whether to attack or retreat, but can communicate only by sending messengers who might never arrive.”

The Godfather

“I stole the idea of the generals and posed the problem in terms of a group of generals, some of whom may be traitors, who have to reach a common decision”

– Lamport on the origin of the Byzantine Generals Problem (1982)

The Story

Our tale begins in Byzantium

Divisions of Byzantine army camped around an enemy city, where each division is commanded by its own general

Generals are tasked with watching the enemy and deciding when to attack

Generals communicate with each other only via messenger

Two options: ATTACK or RETREAT

Problems With The Story

Some generals are traitors!

Treacherous generals try to prevent loyal general from deciding on a common plan, i.e., reaching an agreement

Treacherous generals do whatever they wish, while loyal generals adhere strictly to a clearly defined protocol

Traitors may lie, refuse to communicate, etc.

So What Do We Need?

We need an algorithm that ensures:

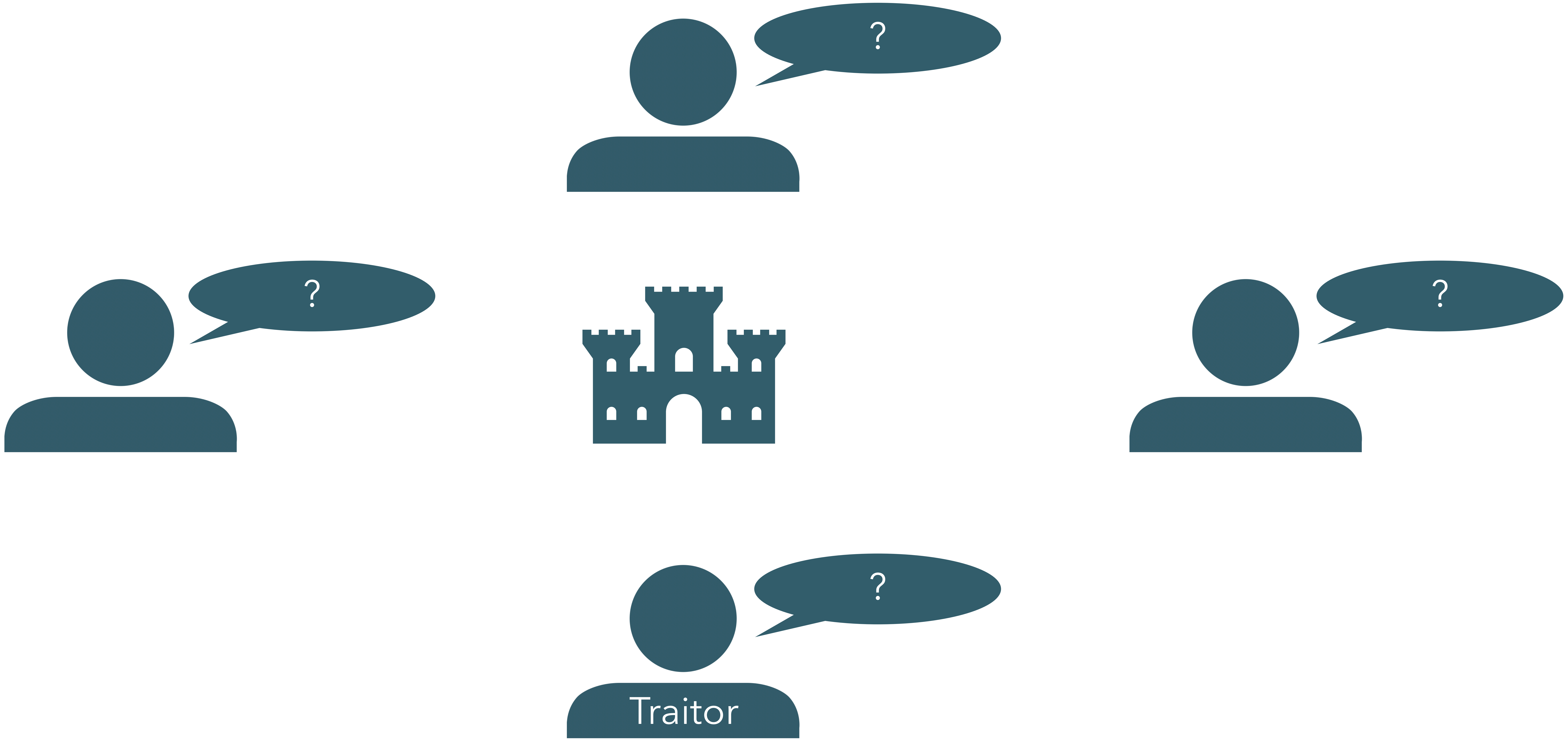
1. All loyal generals decide on the same plan, i.e., they can reach agreement

2. A small number of traitors can not cause loyal generals to adopt a bad plan

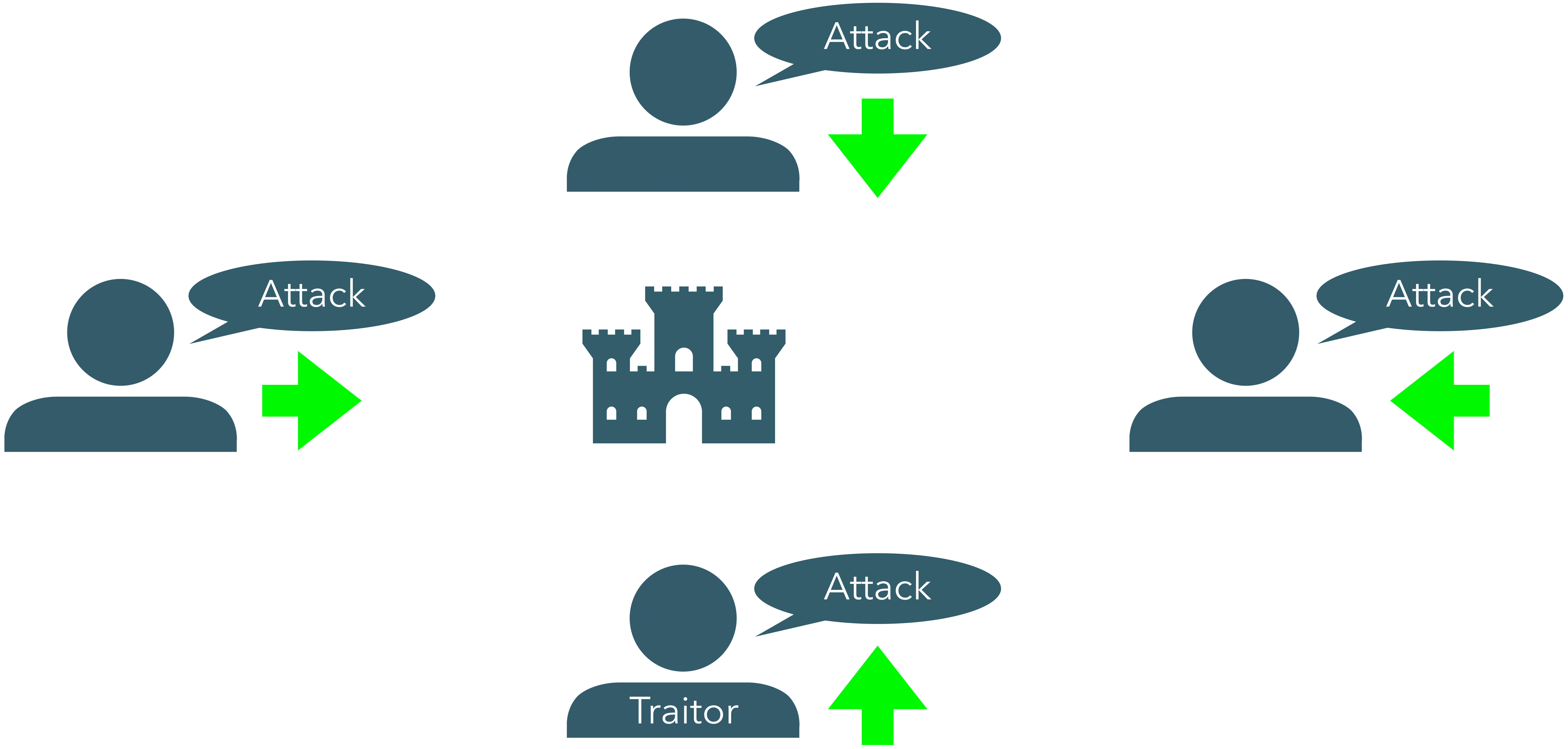
To achieve 1 we need each (loyal) general to have a robust method for combining decisions from other generals

Despite 2 depending on the definition of “bad”, we can achieve it by focusing our efforts on achieving a robust method for 1

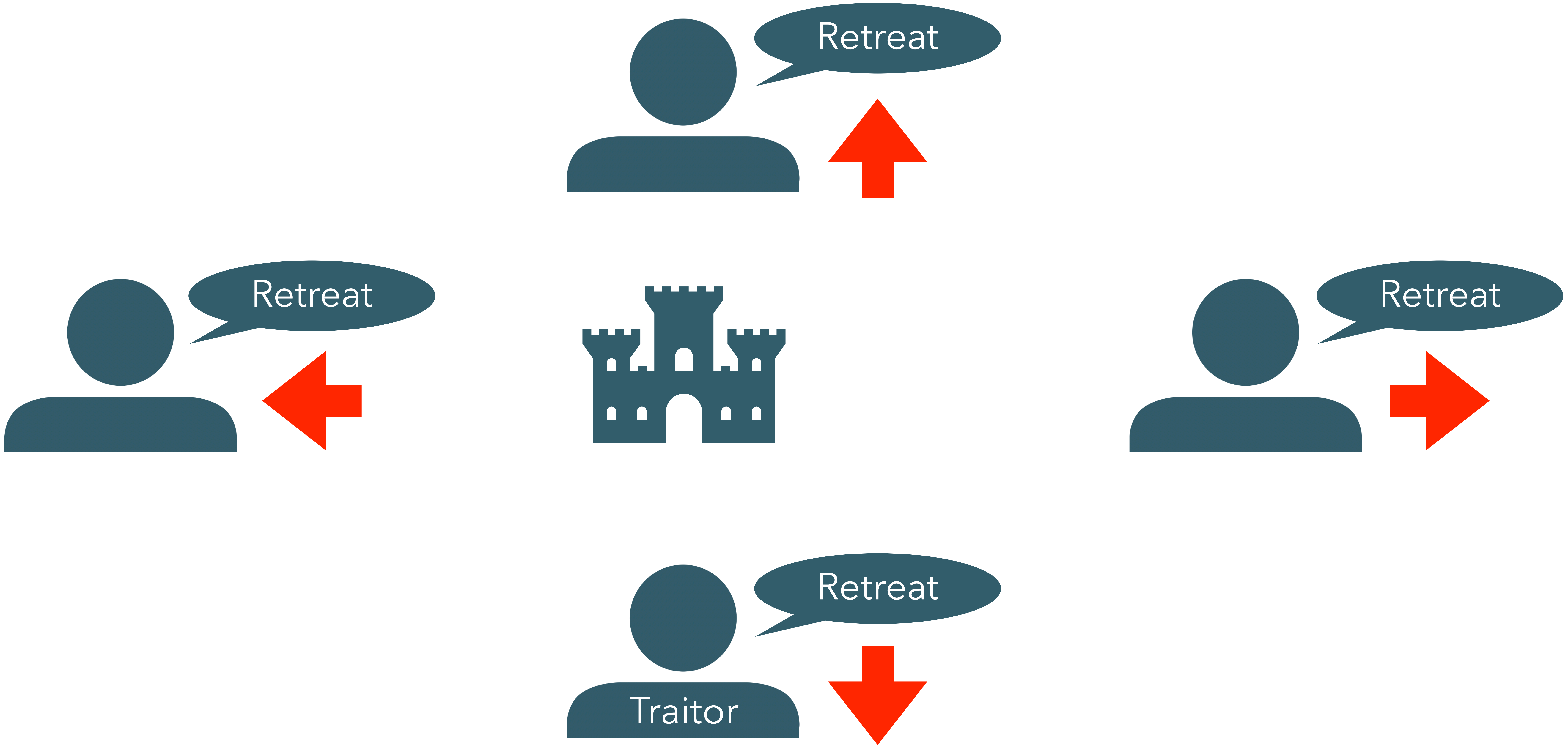
What Is The Byzantine Generals Problem?



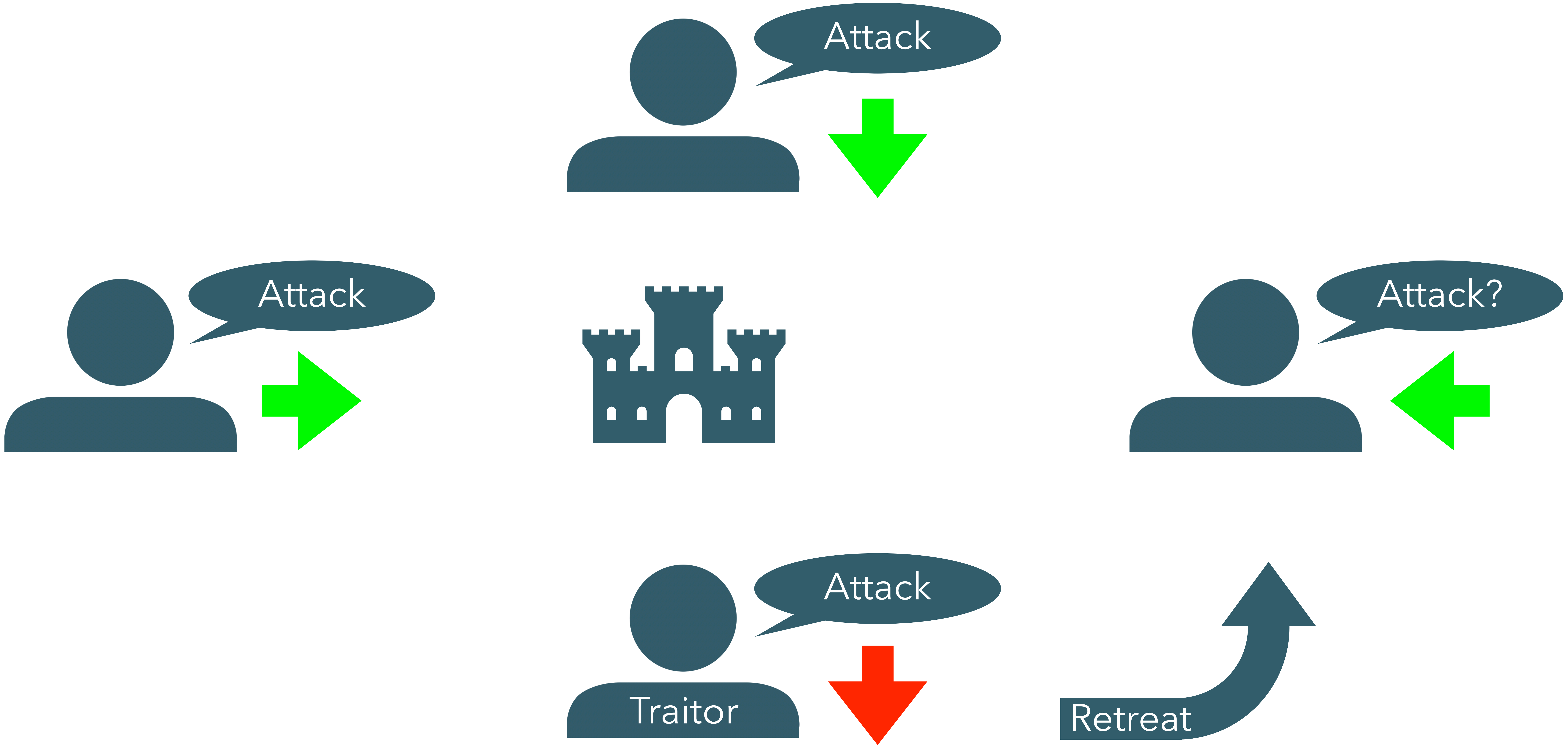
What Is The Byzantine Generals Problem?



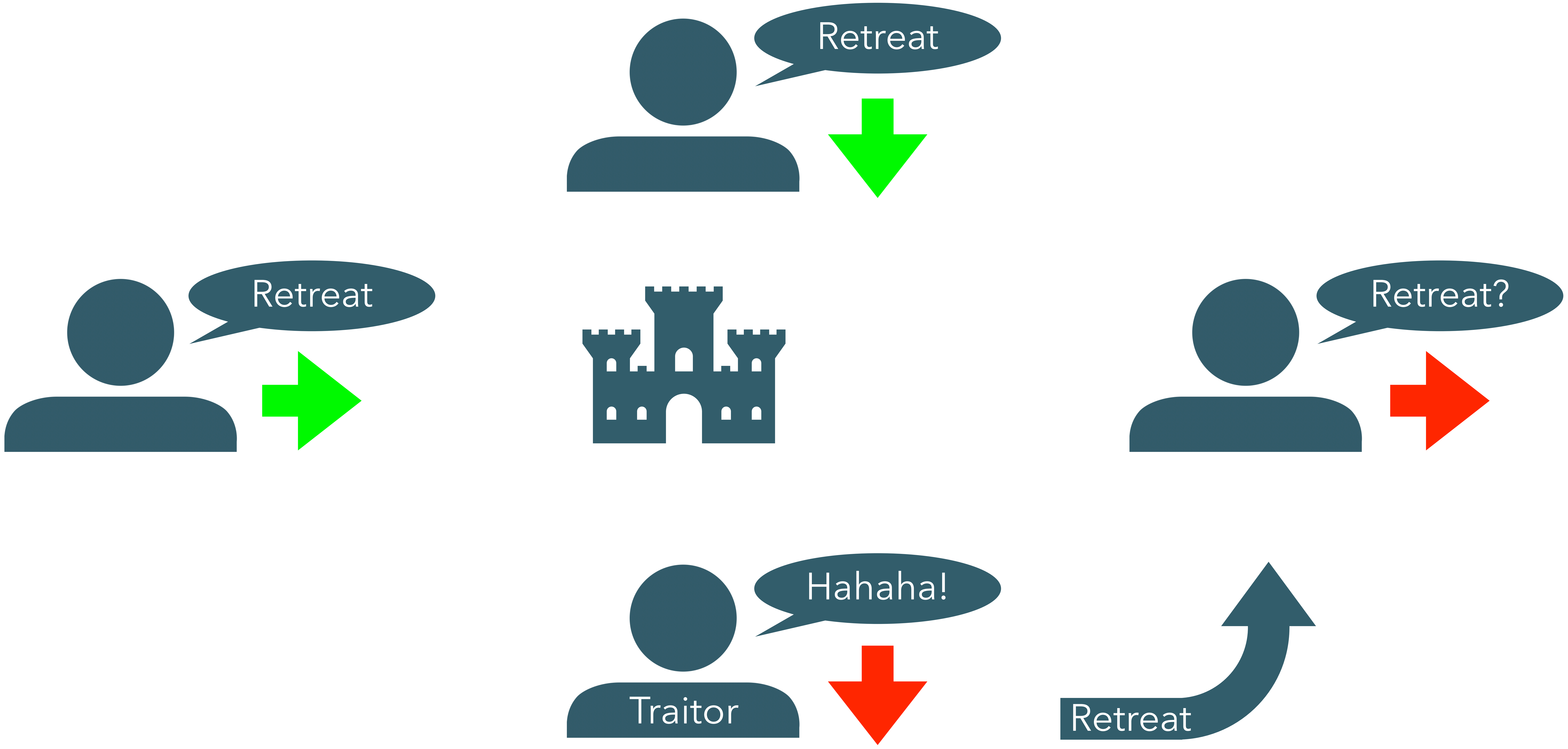
What Is The Byzantine Generals Problem?



What Is The Byzantine Generals Problem?



What Is The Byzantine Generals Problem?



A Simplification

A commanding general sends an order to $n - 1$ lieutenant generals such that:

1. All loyal lieutenants obey the same order

2. If the commanding general is loyal then every loyal lieutenant obeys the order they send

A Simplification - Restated

A commanding general sends an order to $n - 1$ lieutenant generals such that we maintain:

Agreement: All loyal lieutenants obey the same order

Validity: If the commanding general is loyal then every loyal lieutenant obeys the order they send

Termination: Every loyal lieutenants makes a decision on the order they have received.

Impossibility Result

If the generals can send only oral messages, then no solution will work unless more than $\frac{2}{3}$ of the generals are loyal

What are oral messages?

How can we show that this result holds?

Oral Messages

Oral messages have three important properties:

1. Every message sent is delivered correctly

    We saw how critical this is for enabling us to solve solving consensus

2. Receiver of a message knows who sent it

    We refer to this as messages being transmitted in an authenticated channel

3. Message absence can be detected

    We are operating under the synchronous network

Restating The Impossibility Result

If the generals can send only oral messages, then no solution will work unless more than $\frac{2}{3}$ of the generals are loyal

… becomes…

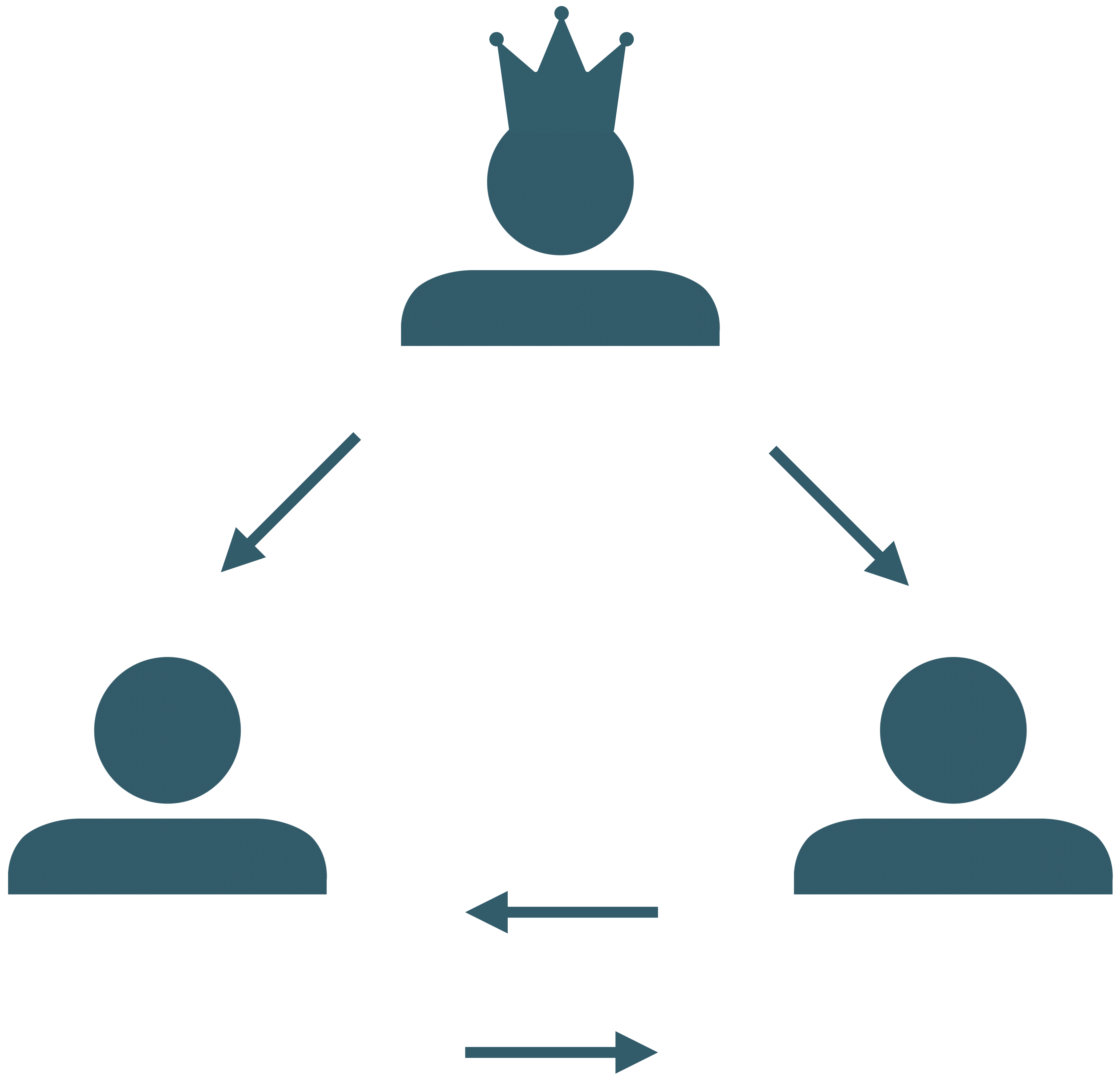
In a synchronous network, with authenticated channel, when m generals are traitors, no solution will work unless there are more than $3m$ generals

The Impossibility Result

The approach to proving the impossibility consists of two steps:

1. Enumerate scenarios for the case where $M = 1$.

2. Assuming we have an algorithm f that works for $3m$ generals when $m > 1$ and identifying a contradiction, hence proving the impossibility



The Impossibility Result

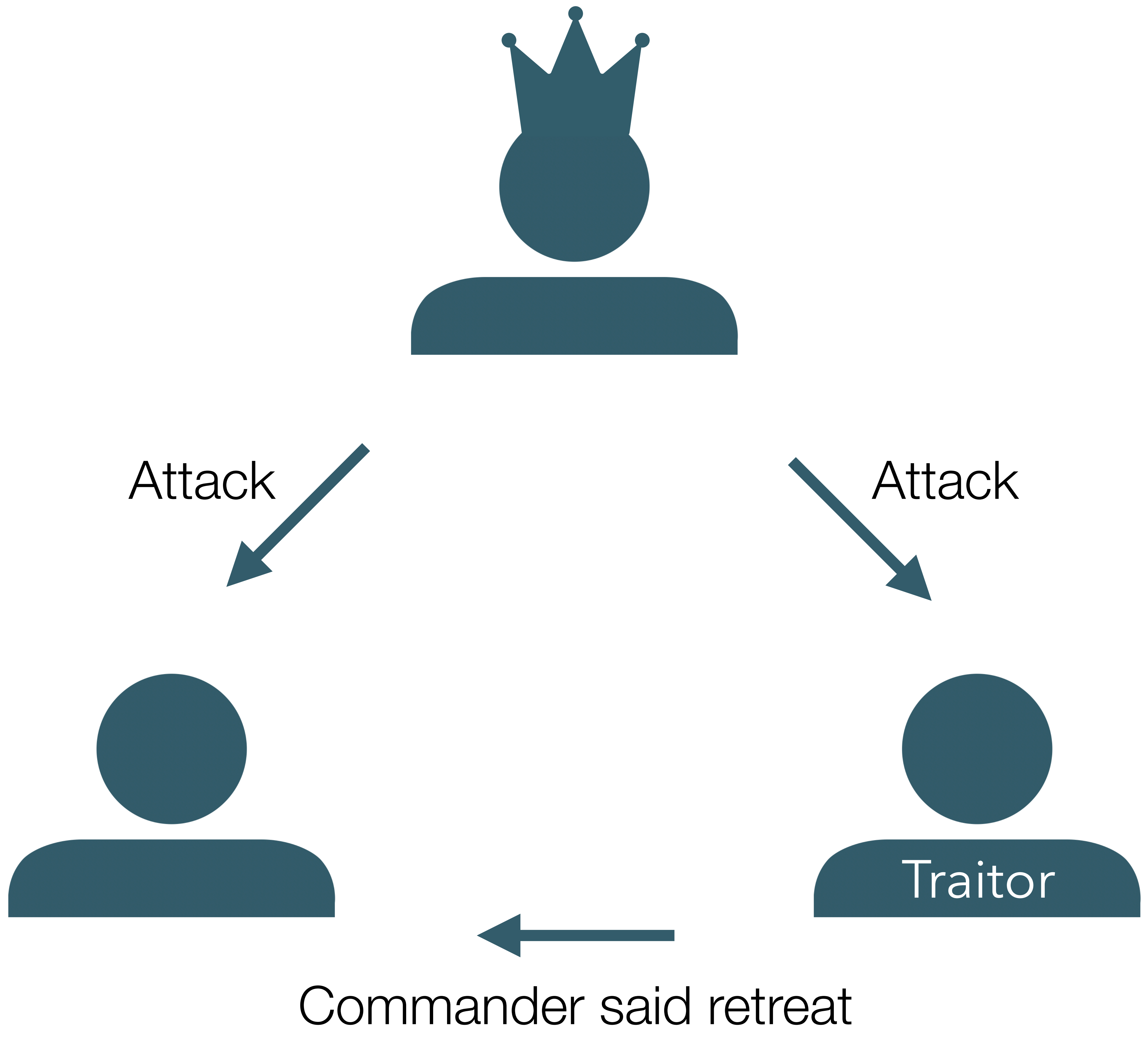
Case $M = 1$

Scenario 1:

Commander is loyal

One lieutenant is a traitor

Non-traitor lieutenant should attack but has cause to mistrust



The Impossibility Result

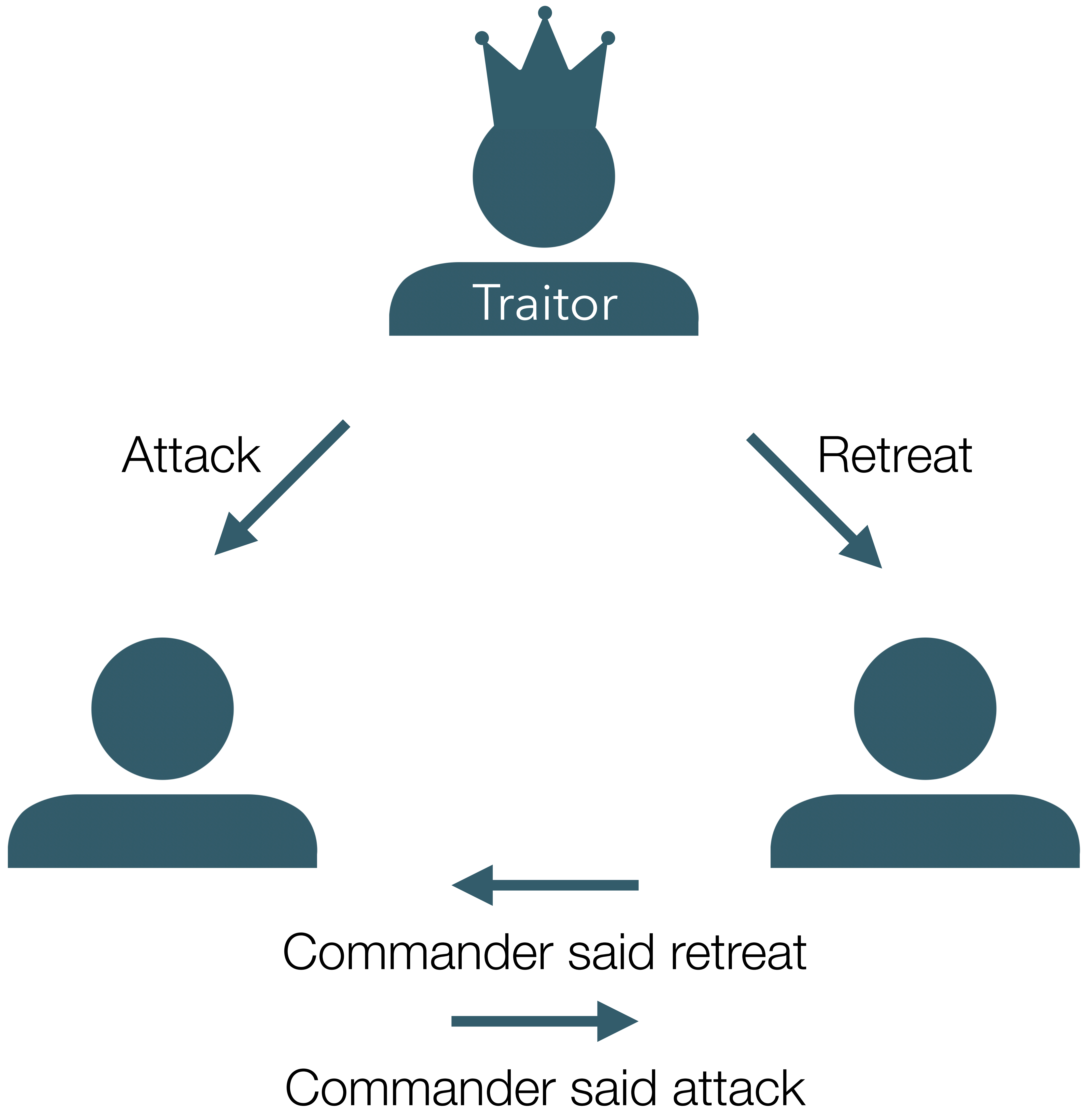
Case $M = 1$

Scenario 2:

Commander is a traitor

Both lieutenants are loyal

Neither lieutenant can agree with each other on the action to take

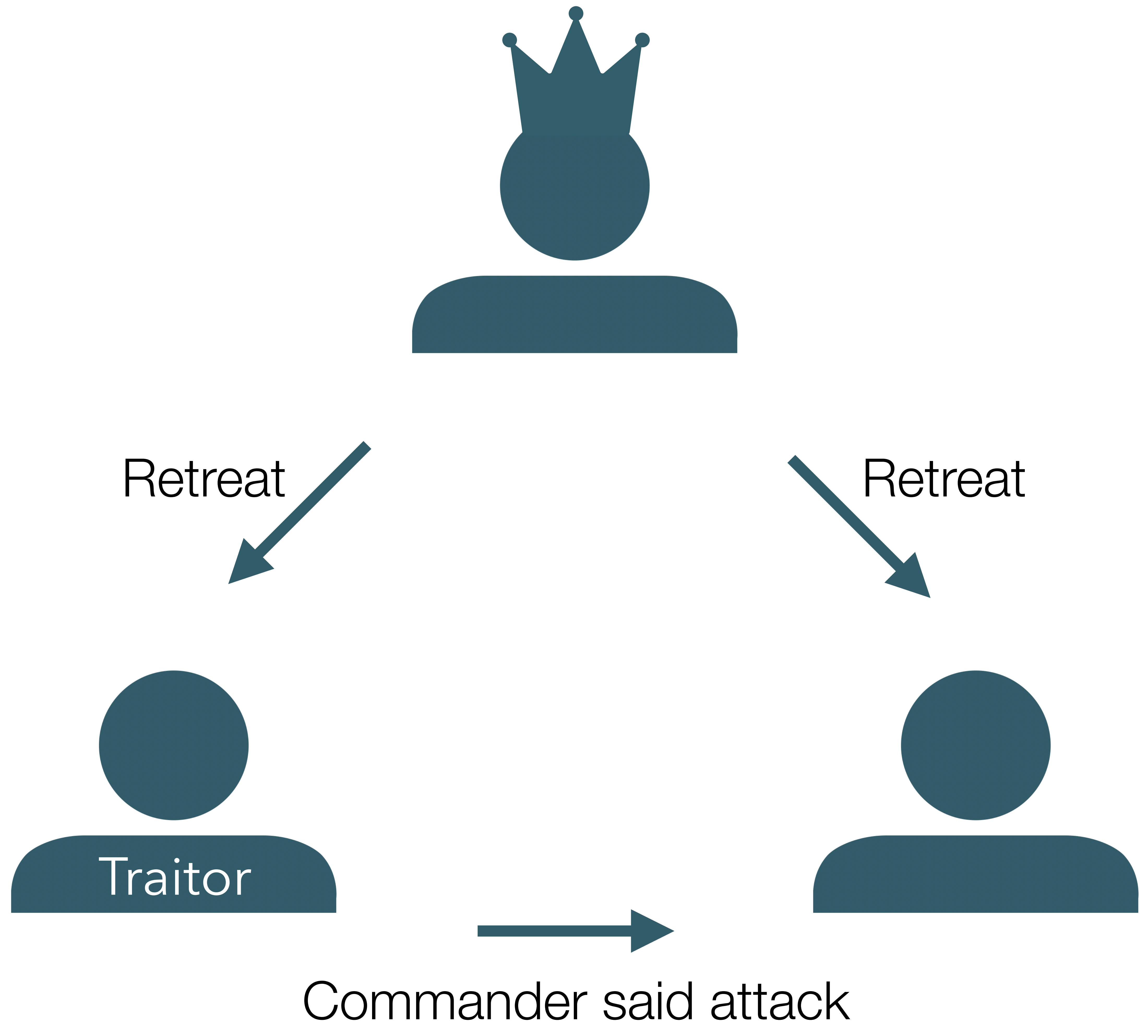


The Impossibility Result

Case $M = 1$

Scenario 3:

- Commander is loyal
- One lieutenant is a traitor (the mirror of Scenario 1)
- Non-traitor lieutenant should retreat but has cause to mistrust



The Impossibility Result

Case $M > 1$

Assume we have an algorithm f that works for $3m$ generals when $m > 1$. We should be able to solve $m = 1$ case by leveraging f.

Labelling the three generals x, y and z, where we take x to be the commander. According to f, x simulates one commander and $m - 1$ lieutenants. Each of y and z simulates m lieutenants.

At most one of x, y and z is a traitor, since we have at most m simulated traitors. Protocol f is assumed to solve the case where we have at most m simulated traitors

We have a contradiction because we have already shown that this is not true for the $m = 1$ case. Hence, there does not exist an algorithm f that works for 3m generals when $m > 1$.

Are Signed Messages A Solution?

Using oral messages, traitors can lie by telling others a command that they never received, a we captured in our worst case analysis

Signed messages can not be forged

Signed messages can be verified as authentic by anyone

Oral Message Solution

Oral Messages Solution to BGP - Algorithm

$OM(k)$:

- $Case\ k = 0$

  Step 1. Commander sends the value to everyone.
  Step 2. Everyone returns the value they received.

- $Case\ k > 0$

  Step 1. Commander sends the value to every one.
  Step 2. Everyone starts a smaller problem $OM(k-1)$ containing all participants but the current commander and becomes the new commander.
  Step 3. everyone participates $n-1$ $OM(k-1)$ and gets $n-1$ values.
  Step 4. Return the majority.

Oral Messages Solution to BGP - Notes

For every message M received, solve a smaller problem containing all but the current commander to tell others that you received M

$$(n-1) * MC(OM(k-1)) + n - 1 = O(n^m)$$

Note that we have $OM(m)$ for m traitors when $3m < n$

Signed Message Solution

Signed Messages Solution to BGP - Algorithm

```
$SM(k)$:

    **Step 1.** Every lieutenant maintains a value set $V(i)$.
    **Step 2.** The commander sends the value to every lieutenant with its signature.
    **Step 3.** For every lieutenant when it receives a new value $v$.
        **Step 3.1.** Put $v$ in $V(i)$
        **Step 3.2.** If $v$ is associated with less than $k$ lieutenants' signatures, sign it and send it to everyone.
    **Step 4.** When there are no more messages to send, return $choose(V)$.

$choose(V)$:

    - Return $v$ when $V = \{v\}$.
    - Return $v_0$ when $|V| = 0$.
```

Signed Messages Solution to BGP - Complexity Analysis

Intuitively we ensure every message received by a loyal lieutenant is sent to every loyal lieutenant

The signed messages solution has a communication complexity of $O(n^2)$

Note that we have $SM(m + 1)$ for m traitors

Practical Byzantine Fault Tolerance

Practical Byzantine Fault Tolerance - High Level Algorithm

$PBFT$:

- Commander sends the value to every lieutenant

- If lieutenant:
    - - Receives a new value v, $broadcast(prepare, v)$
    - - Receives $2f + 1(prepare, v)$, $broadcast(commit, v)$
    - - Receives $2f + 1(commit, v)$, $broadcast(committed, v)$
    - - Receives $f + 1(committed, v)$, $broadcast(committed, v)$

What does this actually mean? Let's consider a simplified algorithm (without corner cases)…

A Simplified PBFT Algorithm

There are n generals, with at least $f+1$ of them are assumed to be loyal.

Each general communicates with other generals through oral messages.

PBFT operates in multiple phases to make a decision: **Pre-Prepare**, **Prepare**, **Commit**, and **Reply**.

PHASE 1 - Pre-Prepare:

General proposes a value (attack or retreat) and broadcasts a pre-prepare message to all generals.

Other generals receive the pre-prepare message and check validity. If valid, move to Prepare phase.

A Simplified PBFT Algorithm

PHASE 2 - Prepare:

On receiving a valid pre-prepare message, a general broadcasts a "prepare" message to all generals.

Generals collect prepare messages from $2f$ other generals (including themselves) for the same value. If they receive enough valid prepare messages, they move to the Commit phase.

PHASE 3 - Commit Phase:

A general, upon receiving $2f$ valid prepare messages, broadcasts a "commit" message to all other generals.

Generals collect commit messages from $2f$ other generals for the same value. If they receive enough valid commit messages, they consider the decision as final.

A Simplified PBFT Algorithm

PHASE 4 - Reply Phase:

Generals reply to the originator of the pre-prepare message with their decision.

Decision:

If a general receives enough commit messages for a particular value, it considers that value as the decided consensus.

Practical Byzantine Fault Tolerance - Notes

Satisfies agreement (what about validity and termination?)

Satisfies liveness under a loyal commander

- What do we do if the commander is a traitor?

Three attractive properties:

- **Transaction finality**: Agreement can be reached without multiple confirmations
- **Energy efficiency**: We can achieve network consensus without requiring energy intensive computation or network traffic
- **Low reward variance**: Collective decision making based on low-votes on records via signing messages

Practical Byzantine Fault Tolerance - Notes

Does not scale well to larger networks because each node must communicated with every other node to keep the network secure

Susceptible to Sybil attacks

- Where a single party creates or manipulates a large number of nodes in the network and compromises security

- Threat less for larger network sizes but considering then we hit the scalability issue

Can be combined (even in-part) with other consensus mechanisms

The f-resiliency Of Consensus Protocols

	Synchronous	Asynchronous	Partially Synchronous
Fail-Stop	f+1	∞	2f+1
Crash	f+1	∞	2f+1 (with Paxos)
Byzantine with digital signatures	f+1 (SM(f+1))	∞	3f+1(PBFT)
Byzantine with authenticated channel	3f+1 (OM(f))	∞	???

The FLP Impossibility

Synchronous, Partially Synchronous and Asynchronous Models

We have studied a variety of algorithms in the context of the synchronous system model

Completely asynchronous models involved steps at arbitrary speeds and in arbitrary orders, which makes the creation of fault tolerant algorithms extremely challenging

> Set of assumptions underpinnings the system model are far less restrictive

Between synchronous and asynchronous network models we have a wide variety of models that can be termed partially synchronous

> For example, processors might have bounds on their relative speeds or might be able to access approximately synchronised clocks

Consensus Properties In Partially Synchronous Networks

Agreement - No two correct processes can decide on different values

Validity - If a process decides on value v then some process must have proposed v

Termination - All processes will eventually decide

Integrity - Each process can decide a value at most once

Why do you think this defines possible solutions in an asynchronous network?

The FLP Impossibility

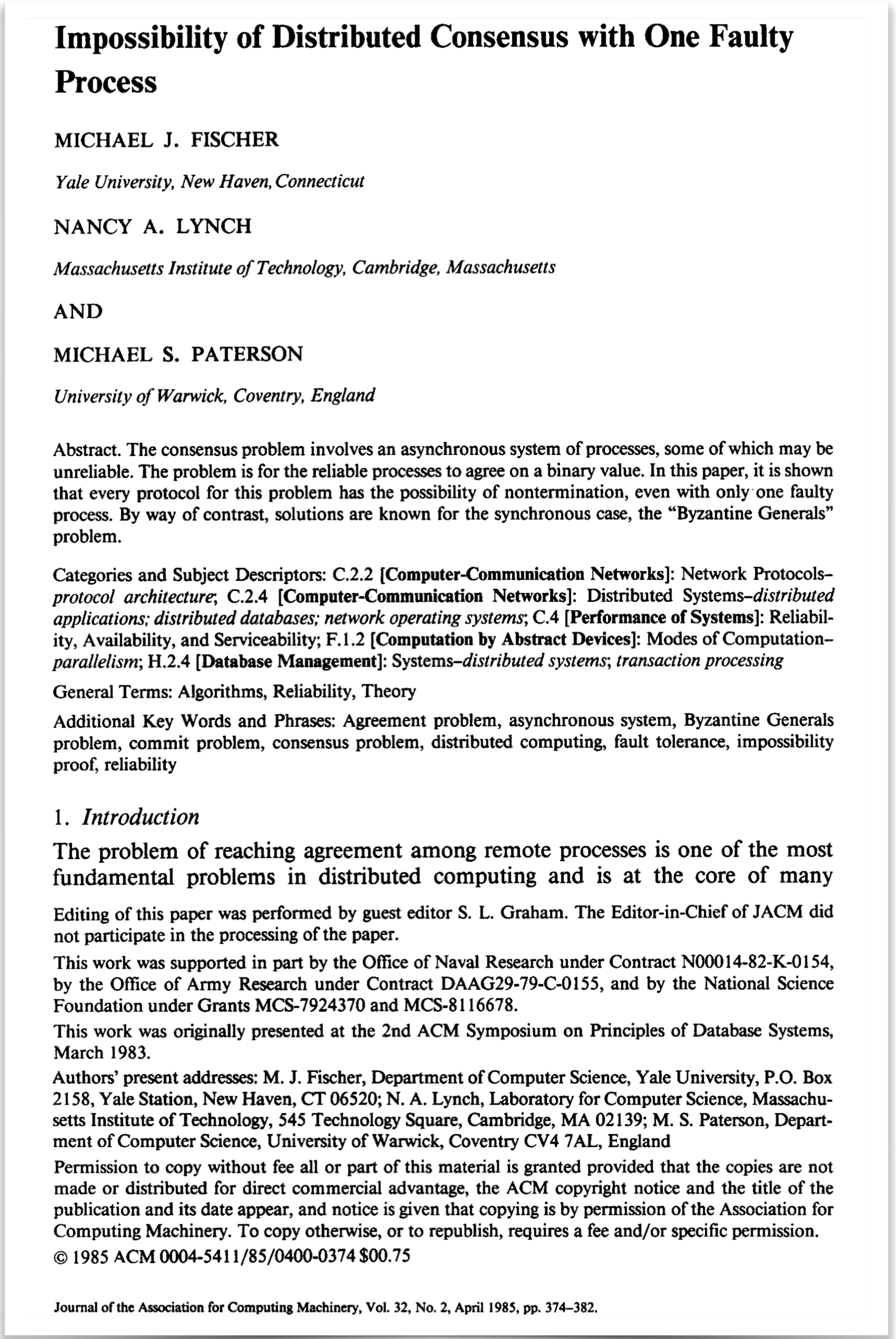
No deterministic one-crash-robust consensus algorithm exists for the asynchronous model

Holds true for "weak" consensus

- Where only some of the processes need to decide

Holds true for only two states

- We know that the binary is not really easier



Impossibility of Distributed Consensus with One Faulty Process

MICHAEL J. FISCHER

Yale University, New Haven, Connecticut

NANCY A. LYNCH

Massachusetts Institute of Technology, Cambridge, Massachusetts

AND

MICHAEL S. PATERSON

University of Warwick, Coventry, England

Abstract. The consensus problem involves an asynchronous system of processes, some of which may be unreliable. The problem is for the reliable processes to agree on a binary value. In this paper, it is shown that every protocol for this problem has the possibility of nontermination, even with only one faulty process. By way of contrast, solutions are known for the synchronous case, the "Byzantine Generals" problem.

Categories and Subject Descriptors: C.2.2 [**Computer-Communication Networks**]: Network Protocols–*protocol architecture*; C.2.4 [**Computer-Communication Networks**]: Distributed Systems–*distributed applications; distributed databases; network operating systems*; C.4 [**Performance of Systems**]: Reliability, Availability, and Serviceability; F.1.2 [**Computation by Abstract Devices**]: Modes of Computation–*parallelism*; H.2.4 [**Database Management**]: Systems–*distributed systems*; *transaction processing*

General Terms: Algorithms, Reliability, Theory

Additional Key Words and Phrases: Agreement problem, asynchronous system, Byzantine Generals problem, commit problem, consensus problem, distributed computing, fault tolerance, impossibility proof, reliability

1. *Introduction*

The problem of reaching agreement among remote processes is one of the most fundamental problems in distributed computing and is at the core of many

Editing of this paper was performed by guest editor S. L. Graham. The Editor-in-Chief of JACM did not participate in the processing of the paper.

This work was supported in part by the Office of Naval Research under Contract N00014-82-K-0154, by the Office of Army Research under Contract DAAG29-79-C-0155, and by the National Science Foundation under Grants MCS-7924370 and MCS-8116678.

This work was originally presented at the 2nd ACM Symposium on Principles of Database Systems, March 1983.

Authors' present addresses: M. J. Fischer, Department of Computer Science, Yale University, P.O. Box 2158, Yale Station, New Haven, CT 06520; N. A. Lynch, Laboratory for Computer Science, Massachusetts Institute of Technology, 545 Technology Square, Cambridge, MA 02139; M. S. Paterson, Department of Computer Science, University of Warwick, Coventry CV4 7AL, England

Journal of the Association for Computing Machinery, Vol. 32, No. 2, April 1985, pp. 374–382.

The FLP Impossibility

We consider configurations of a system that can ultimately can lead to a 0 or 1 result

The states of five processes characterise a single configuration

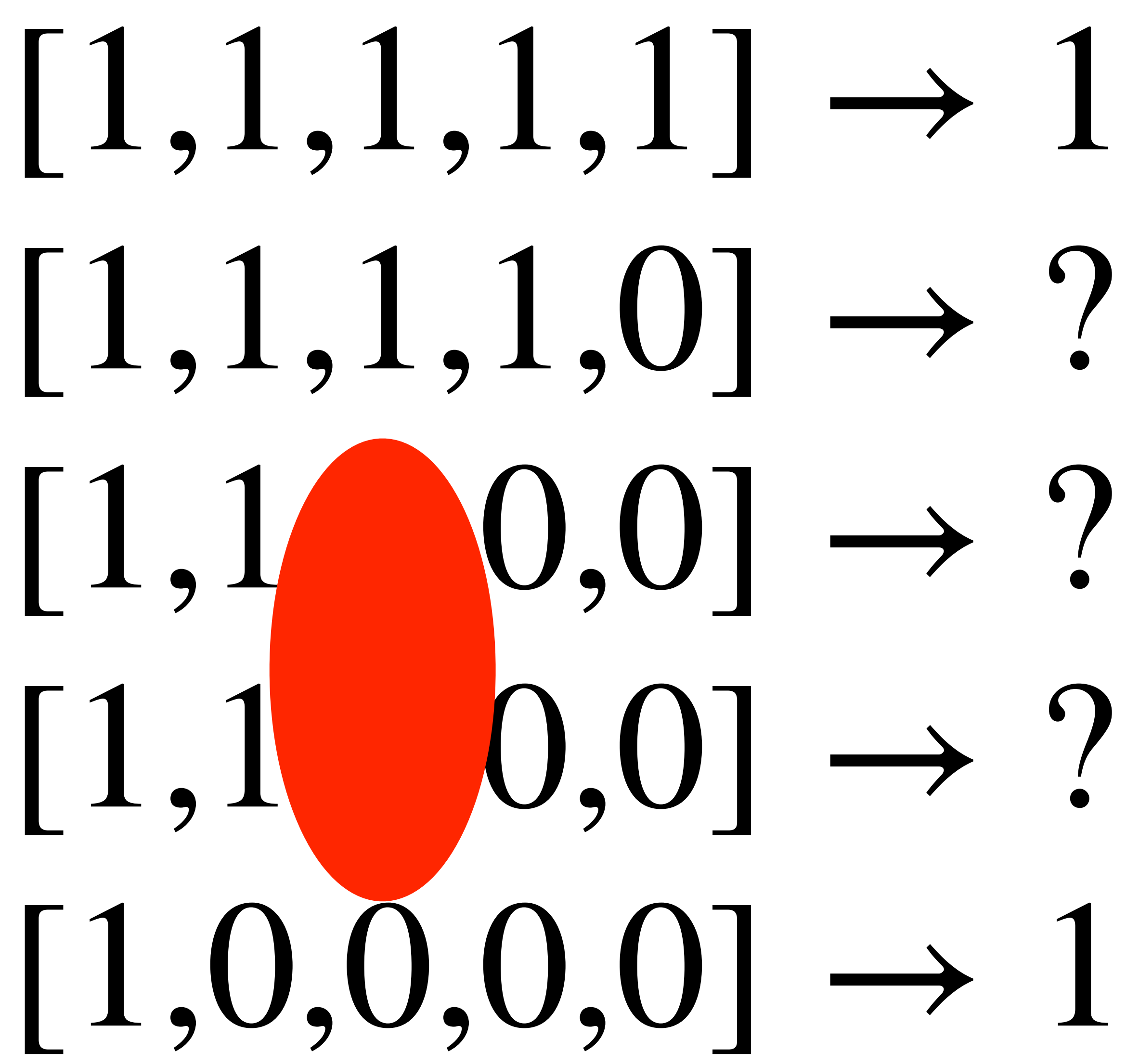
There must exists two configurations of the network whereby a single failure would result in an altered outcome

- Imagine hiding the state of the failed process in those configurations

- One of these configurations must be “bi-valent”, meaning that both possible decisions are possible in the context of failure

Inherent non-determinism from our asynchronous network

Bi-valent states can remain in bi-valent states after performing some work



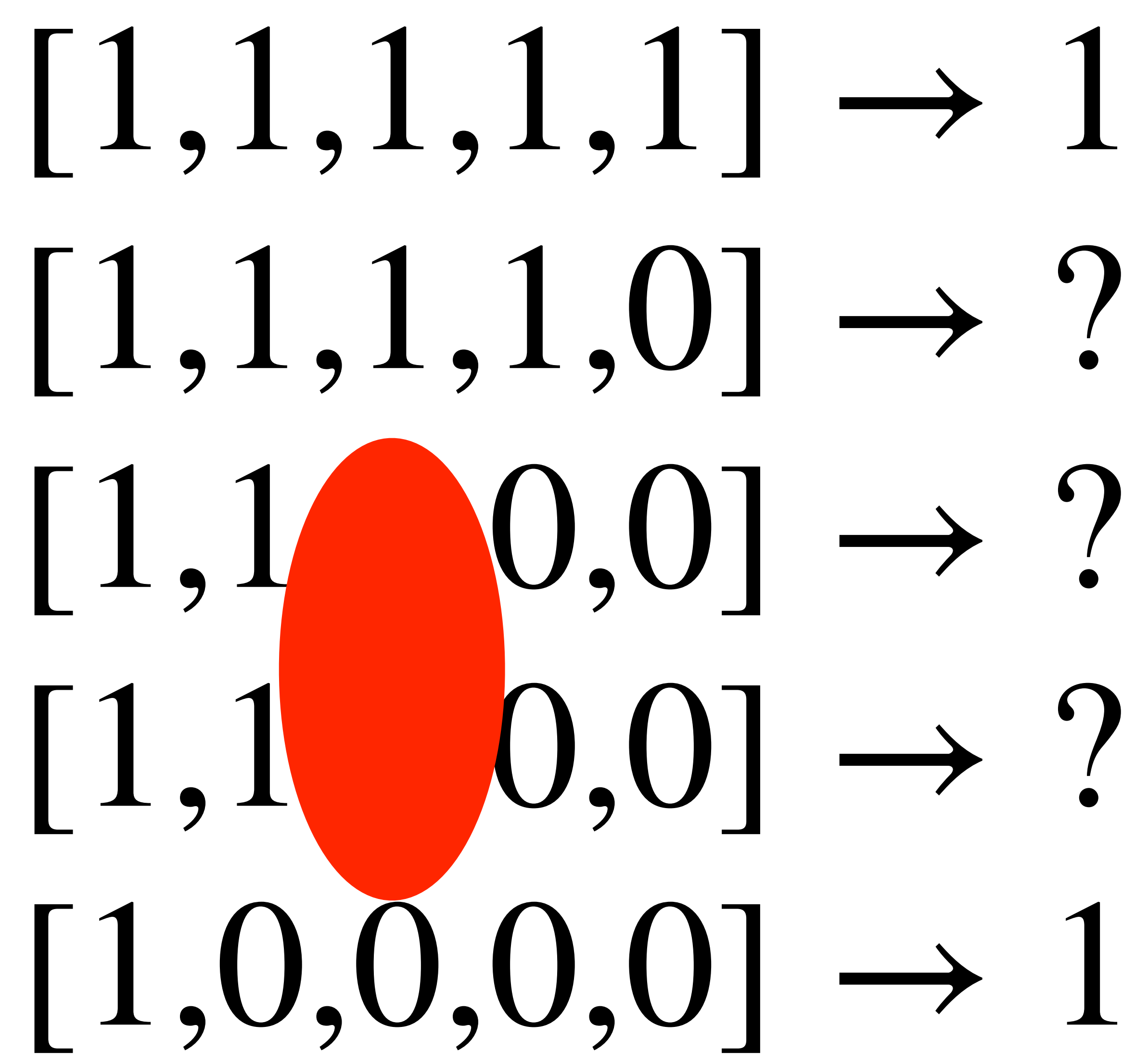
The FLP Impossibility

Impossibility means we can not simultaneously provide safety, liveness, asynchrony and fault tolerance

In an asynchronous network we can design for safety and liveness but we can’t tolerate faults

If we want to provide fault tolerance then we would have to sacrifice (at least one of) safety, liveness or asynchrony

Impossibility does NOT mean that the distributed consensus problem can not be solved in general



Paxos

Paxos

Paxos is a consensus algorithm that can be used in a partially synchronous network

We need to think about what this actually means for the assumptions we make

One or more clients proposes a value and we have consensus when a majority of systems running Paxos agrees on one of the proposed values

Paxos was the first consensus algorithm that has been rigorously proved to be correct

Leslie Lamport is back!!!

Paxos

Paxos selects a single value from one or more values that are proposed to it and lets everyone know that value

A single run of the Paxos protocol results in the selection of single proposed value

If we need to use Paxos to create a replicated log, e.g., for a replicated state machine, then we need to run Paxos repeatedly

Multiple execution of the base Paxos protocol is known are multi-Paxos

Paxos

Paxos provides abortable consensus

> This means that some processes abort the consensus if there is contention while others decide on the value

Those processes that decide have to agree on the same value

Aborting allows a process to terminate rather than be blocked indefinitely

When a client proposes a value to Paxos, it is possible that the proposed value might fail if there was a competing concurrent proposal that won

The client will then have to propose the value again to another run of the Paxos algorithm

Paxos - Assumptions

Concurrent Proposals

One or more systems may propose a value concurrently

If only one system would propose a value then it is clear what the consensus would be, whilst multiple systems need to select one from among those values

Validity

The chosen value that is agreed upon must be one of the proposed values

Paxos - Assumptions

Majority Rule

Once a majority of Paxos servers agrees on one of the proposed values, we have consensus on that value

This also implies that a majority of servers need to be functioning for the algorithm to run

To survive m failures, we will need $2m + 1$ systems.

Asynchronous Network

The network is unreliable and asynchronous, hence messages may be lost or delayed

Paxos - Assumptions

Process Failure

Systems may exhibit process failures (not initially Byzantine) whereby the system must need restart or remember their previous state to make sure they do not change their mind

Unicasts

Communication is point-to-point, implying no mechanism to multicast a message atomically to a set of Paxos servers

Announcement

Once consensus is reached, the results can be made known to everyone

Paxos - Outline

Operates in asynchronous networks, i.e., no bounds on message timings or process operating speed

The algorithm ensures safety and liveness

Uses **proposers**, **acceptors** and **learners**

Proposer

Proposers are responsible for initiating the consensus process by suggesting a value. They send a prepare message to a majority of acceptors, suggesting a value and asking them to promise not to accept any proposals numbered lower than the one in the prepare message.

Paxos - Outline

Acceptor

Acceptors receive prepare messages and respond with promises not to accept any proposal numbered lower than the one in the prepare message. If the acceptor has already accepted a proposal, it includes the proposal's number and value in its response.

Learner

Learners are the nodes that eventually learn the decided value. They gather enough information from acceptors and reach a consensus on the agreed-upon value.

Paxos - Outline

We assume an environment of multiple systems connected by an partially synchronous or asynchronous network, where one or more of the system may concurrently propose a value

Proposing a value might correspond to performing an operation on a server, selecting a coordinator, adding to a log, or some other activity that requires consistent across systems

We need a protocol to choose exactly one value in cases where multiple competing values may be proposed

Paxos - Outline

Prepare Phase

Proposers send prepare messages to a majority of acceptors.

Acceptors respond with promises not to accept any proposals lower than the one in the message.

Accept Phase

If a proposer receives promises from a majority of acceptors, it can send an accept message to the acceptors, including the proposed value.

Acceptors, upon receiving an accept message, accept the proposal unless they have already promised not to accept a proposal with a lower number.

Paxos - Outline

Learn Phase

Once a proposal is accepted, the learner can gather enough acceptances to determine the decided value.

Implementing Paxos directly is quite challenging, since there are nuances

Useful to work through the notions of phrases as expressed here alongside the Paxos notes and Full Paxos implementation

Paxos - Multiple Proposers, Single Acceptor

Simplest design uses a single acceptor

We can choose one of our systems to take on the role of the acceptor

Multiple proposers will concurrently propose various values to a single acceptor

Unfortunately, this does not handle the case where the acceptor crashes.

- If the acceptor crashes after choosing a value, we will not know what value has been chosen and will have to wait for the acceptor to restart
- Redundancy helps us to address the fault tolerance issues

Paxos - Multiple Proposers, Multiple Acceptors

Use a collection of acceptors

Each proposal will be sent to at least a majority of these acceptors

If a quorum (typically majority) of these acceptors chooses a particular proposed value (proposal) then that value will be considered chosen

Even if some acceptors crash, others are up so we will still know the outcome

If an acceptor crashes after it accepted a proposal, other acceptors know a value was chosen

Paxos - Algorithm Notes

A client sends a request to any Paxos proposer, who runs a two-phase protocol with the acceptors

Paxos is a majority-win protocol

- Avoids split-brain problems to ensure that, if you make a proposal asking over 50% of the systems if somebody else made a proposal and they all reply negatively, you know for certain that no other system could have asked over 50% of the system regarding a commensurate proposal

- Paxos requires a majority of its servers to be running for the algorithm to terminate

- Majority ensures at least one node in common from one majority to another if servers fail and restart

- The system requires $2m + 1$ servers to tolerate the failure of m servers

Paxos - Algorithm Notes

When a proposer receives a client request to reach consensus on a value, the proposer must create a proposal number that satisfies two properties:

1. **Uniqueness** - No two proposers can come up with the same number

    An easy way of doing this is to use a global process identifier for the least significant bits of the number, e.g., instead of $ID = 12$, Node 3 will generate $ID = 12.3$ and node 2 will generate $ID = 12.2$, etc.

2. **Ascending** - The number be bigger than any previously used identifier used in the cluster

    A proposer may use an incrementing counter or use a nanosecond-level timestamp to achieve this

    If the number is not bigger than one previously used, the proposer will find out by having its proposal rejected and will have to try again

Full Paxos - Phase 1 Proposer (PREPARE)

A proposer initiates a $PREPARE$ message, picking a unique, ever-incrementing value

```
$ID = count++$
$send(PREPARE)$
```

Full Paxos - Phase 1 Acceptor (PROMISE)

An acceptor receives a $PREPARE(ID)$ message:

```
$if\ ID \leq maxKnownID$
    $reply\ FAIL$
$else$
    $maxKnownID$
    $if\ proposalAccepted$
            $reply\ PROMISE(ID, AcceptedID, AcceptedValue)$
    $else$
            $reply\ PROMISE(ID)$
```

Full Paxos - Phase 2 Proposer (PROPOSE)

The proposer now checks to see if it can use its proposal or if it has to use the highest one it has received from among all responses

```
$if\ responsesRecieved > (0.5 * acceptorTotal)$
    $if\ responsesContain(acceptedValues)$
            $val = acceptedValue$
    $else$
            $val = value$
    $send\ PROPOSE(ID, val)$
```

Full Paxos - Phase 2 Acceptor (ACCEPT)

Each acceptor receives a $PROPOSE(ID, VALUE)$ message from a proposer

If the ID is the highest number it has processed then accept the proposal and propagate the value to the proposer and to all the learners

```
$if\ ID == maxKnownID$
    $proposalAccepted = TRUE$
    $acceptedID = ID$
    $acceptedValue = value$
    $reply\ ACCEPTED(ID, value)$
$else$
    $reply\ FAIL$
```

Failure Cases - Does Full Paxos Work?

Acceptor failure in Phase 1

Acceptor will not return a *PROMISE* message but, as long as the proposer still gets responses from a majority of acceptors, the protocol can continue to make progress

Acceptor failure in Phase 2

Acceptor will not be able to send back an *ACCEPTED* message but this is not a problem if enough acceptors are still alive and will respond so that the proposer or learner receives responses from a majority of acceptors

Failure Cases - Does Full Paxos Work?

Proposer failure in Phase 1

Depends on the pattern of messages that has taken place, hence we have three cases:

1. Proposer failing before it sent any messages is equivalent to it never having run

2. Proposer failing after sending one or more $PREPARE$ messages means an acceptor would have sent PROMISE responses back but no $ACCEPT$ messages would follow, hence another node will run Paxos as a proposer (picking its own ID)

3. If at least one $ACCEPT$ message was sent then some another node proposes a new message with $PREPARE(higherID)$, with the acceptor responding by telling that proposer that an earlier proposal was already accepted

Failure Cases - Does Full Paxos Work?

Proposer failure in Phase 2:

A proposer that takes over does not know it is taking over a pending consensus, it just proposes a value, giving us two cases to consider for Phase 2:

1. The proposer does not get any responses from a majority of acceptors that contain an old proposal ID and corresponding value, meaning there has been no majority agreement and the protocol can run in full

2. The proposer that takes over knows it is taking over a pending consensus because it gets at least one response that contains an accepted proposal number and value, hence it executes using that previous value and completes the protocol

Paxos and Livelock

Livelock - Two or more processes continually repeat some interaction in response to changes in the other processes without doing any useful work

Distinct from deadlock because in a deadlock all processes are in some form of waiting state

Similar to a deadlock, except that the states of the processes involved in the livelock constantly change with regard to one another, none progressing

Considered a special case of process starvation in the context of operating systems

Resolving Livelock in Paxos (Leader Election Strikes Back)

Possible to use random exponential (increasing) delays

More common to select a single proposer as a leader to manage all incoming events

> We can run Paxos to select a leader but, to ensure that we do not encounter livelock while running leader election, an alternative is a better choice
>
> Commercial users often use the Bully algorithm to choose a leader

Bully algorithm - A node starts an election by sending its identifier to all peers. If it gets a response from any peer with a higher identifier, it will not be the leader. If all responses have lower identifiers then it becomes the leader. If a node receives an election message from a node with a lower identifier, the node starts its own election. The node with the highest identifier will be the winner.

Real World Paxos

Single Run or multi-run

Paxos yields consensus for a single value

Much of the time, we need to make consensus decisions repeatedly, such as keeping a replicated log or state machine synchronized, hence we turn to multi-Paxos

Byzantine failures

In design we assumed that none of the systems running Paxos suffer Byzantine failures

We can guard against network problems with mechanisms such as checksums or digital signatures but we do need to worry about a misbehaving proposer that may inadvertently set its proposal ID to infinity

This puts the Paxos protocol into a state where acceptors will have to reject any other proposal

Real World Paxos

Location of servers

The most common use of Paxos is in implementing replicated state machines, such as a distributed storage system

To ensure that replicas are consistent, incoming operations must be processed in the same order on all systems

A common way of doing this is to use a replicated log, whereby each of the servers will maintain a log that is sequenced identically to the logs on the other servers

A consensus algorithm will decide the next value that goes on the log so that each server only has to process log in order and apply the requested operations

Real World Paxos

Group administration

The cluster of systems that are running Paxos needs to be administered

We need to be able to add systems to the group, remove them, and detect if any processes, entire systems, or network links are dead

Each proposer needs to know the set of acceptors so it can communicate with them and needs to know the learners (if those are present)

Paxos is a fault-tolerant protocol but the mechanisms for managing the group are outside of its scope

The f-resiliency Of Consensus Protocols

	Synchronous	Asynchronous	Partially Synchronous
Fail-Stop	f+1	∞	2f+1
Crash	f+1	∞	2f+1 (with Paxos)
Byzantine with digital signatures	f+1 (SM(f+1))	∞	3f+1(PBFT)
Byzantine with authenticated channel	3f+1 (OM(f))	∞	???