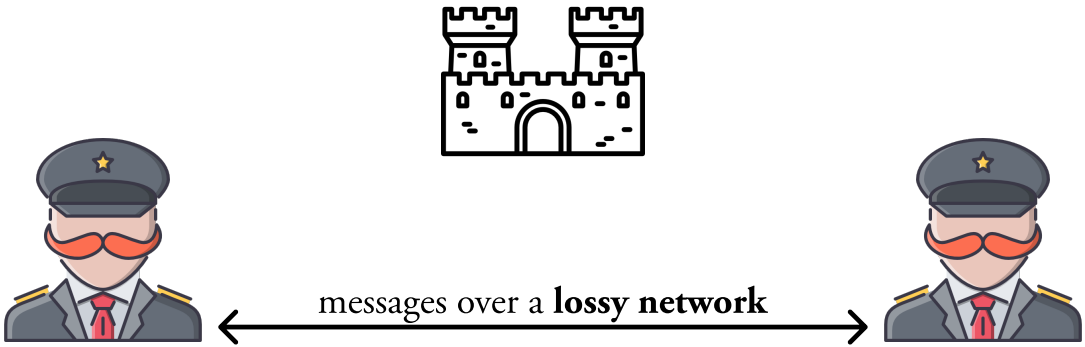
Logical models

Generals problem

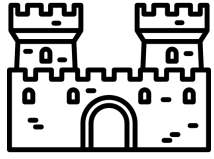


General 1	General 2	Outcome
Do not attack	*Do not attack*	*Nothing happens*
Attack	*Do not attack*	*General 1 defeated*
Do not attack	*Attack*	*General 2 defeated*
Attack	*Attack*	*Fortress captured*

Goal:

General 1 attacks
if and only if
General 2 attacks

Generals problem



General 1

General 2

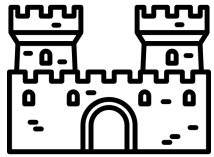
No common knowledge:

only way of knowing something
is to communicate it

Goal:

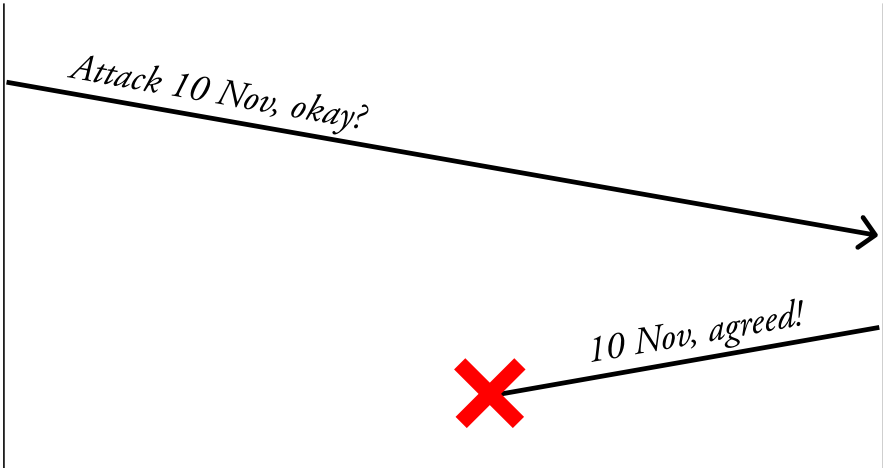
General 1 attacks
<u>if and only if</u>
General 2 attacks

Generals problem



General 1

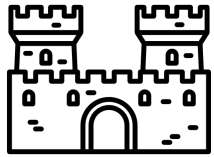
General 2



Goal:

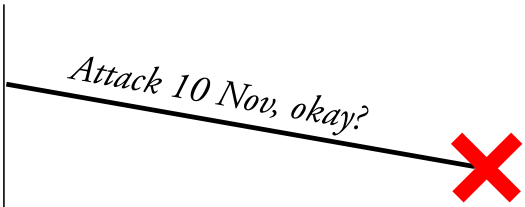
General 1 attacks
if and only if
General 2 attacks

Generals problem



General 1

General 2



Goal:

General 1 attacks
if and only if
General 2 attacks

Option 1:

I attack anyway... but I am not safe!
Lot of mugs increase probability
that General 2 eventually knows.

Option 2:

I wait for commitment of General 2.
At least I am safe...

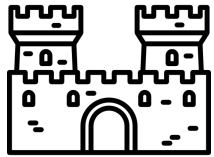
General 1

General 2

Option 1':
I always commit to attack... but I am not safe!
Lot of mugs increase probability
that General 1 eventually knows.

Option 2':
I never agree to attack.
At least I am safe...

The two generals problem in practice



Order

messages / RPC

Online shop

"Dispatch item?"

Payment service

"Charge credit card?"

Online shop	Payment service	Outcome
Do not dispatch	*Do not charge*	*Nothing happens*
Dispatch	*Do not charge*	*Shop loses*
Do not dispatch	*Charge*	*Customer unhappy*
Dispatch	*Charge*	*Everyone happy*

Goal:

Online shop dispatches
if and only if
Payment service charges

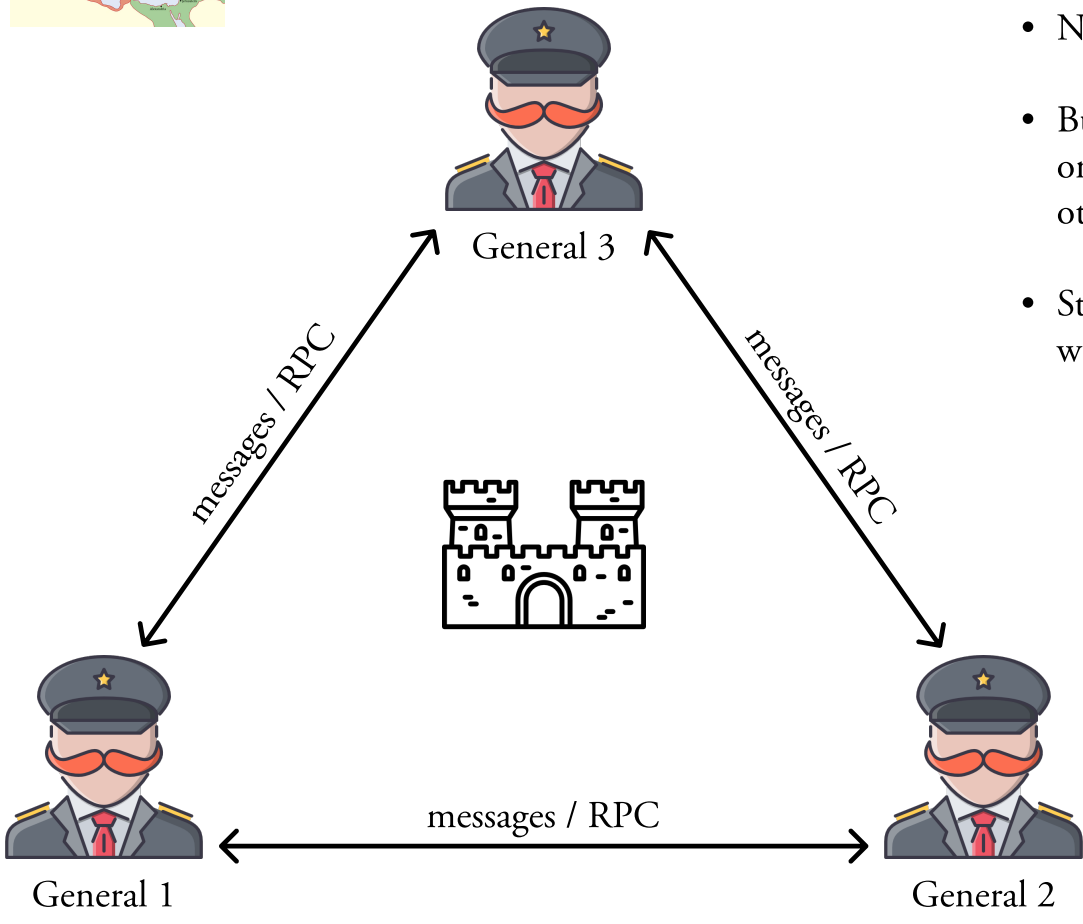
Byzantine generals problem

Byzantine

"Byzantine" has long been used for "excessively complicated, bureaucratic, nor worth understanding..."

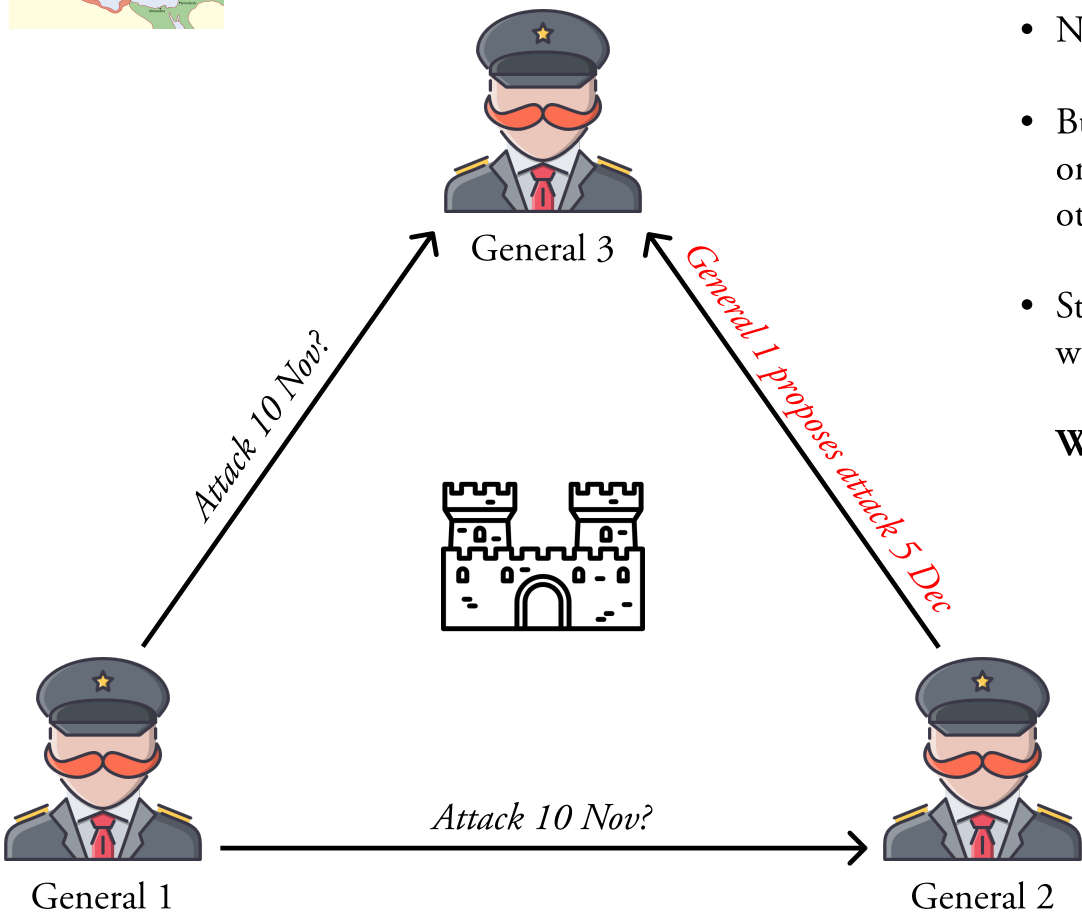
Byzantium / Constantinople / Istanbul

Byzantine generals problem



- Network is now reliable
- But generals are not:
  one general is **malicious**,
  other generals don't know who it is
- Still, honest generals
  would like to agree on a plan

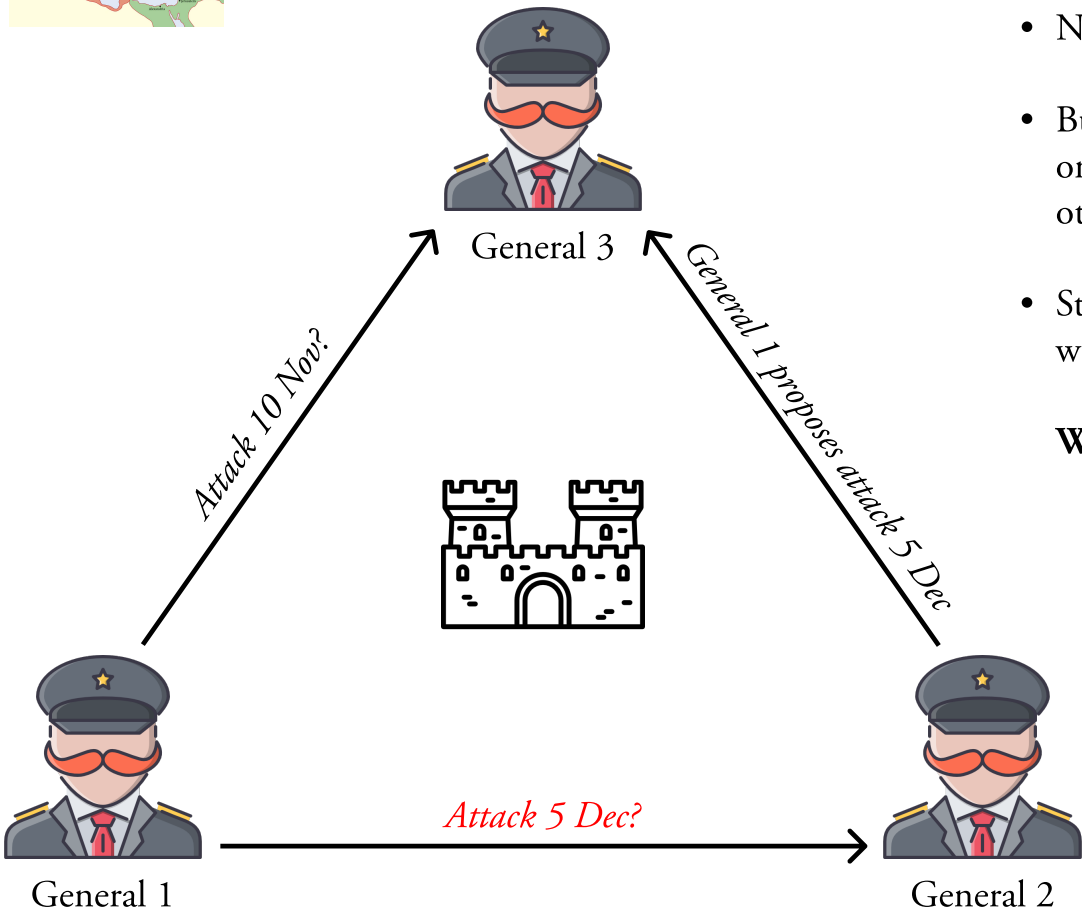
Byzantine generals problem



- Network is now reliable
- But generals are not:
  one general is **malicious**,
  other generals don't know who it is
- Still, honest generals
  would like to agree on a plan

Who can trust whom?

Byzantine generals problem



- Network is now reliable
- But generals are not:
  one general is **malicious**,
  other generals don't know who it is
- Still, honest generals
  would like to agree on a plan

  Who can trust whom?

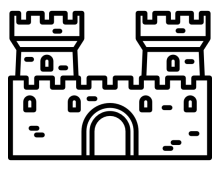
Byzantine generals problem in practice!

Customer

Buy item?

messages / RPC

messages / RPC



Order

messages / RPC

Online shop

Dispatch item?

Payment service

Charge card?

- Network is now reliable
- But generals are not:
  one general is **malicious**,
  other generals don't know who it is
- Still, honest generals
  would like to agree on a plan

Who can trust whom?

Faulty behaviours

We have seen two thought experiments:

- **Two generals problem**: a model of *faulty networks*
- **Byzantine generals problem**: a model of *faulty node behaviour*

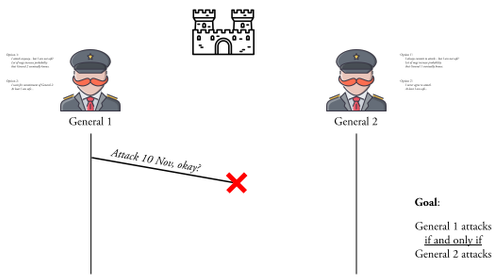
In real systems, both nodes and networks may be faulty!

A generic (failure-oriented) **logical model** of a distributed system may capture:

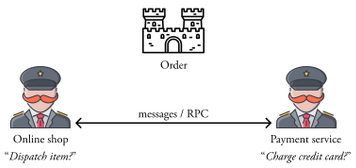
- *Network behaviour* (e.g. message loss, reordering, duplication, ...)
- *Node behaviour* (e.g. crashes, malicious behaviours, ...)
- *Timing guarantees* (e.g. latency, execution time, clock drifting, ...)

Logical models

Generals problem



The two generals problem in practice

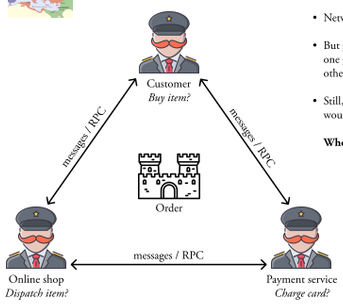


Online shop	Payment service	Outcome
Do not dispatch	*Do not charge*	*Nothing happens*
Dispatch	*Do not charge*	*Shop loses*
Do not dispatch	*Charge*	*Customer unhappy*
Dispatch	*Charge*	*Everyone happy*

Goal:

Online shop dispatches
if and only if
Payment service charges

Byzantine generals problem in practice!



- Network is now reliable
- But generals are not:
  one general is **malicious**,
  other generals don't know who it is
- Still, honest generals
  would like to agree on a plan

Who can trust whom?

Faulty behaviours

We have seen two thought experiments:

- **Two generals problem**: a model of *faulty networks*
- **Byzantine generals problem**: a model of *faulty node behaviour*

In real systems, both nodes and networks may be faulty!

A generic (failure-oriented) **logical model** of a distributed system may capture:

- *Network behaviour* (e.g. message loss, reordering, duplication, ...)
- *Node behaviour* (e.g. crashes, malicious behaviours, ...)
- *Timing guarantees* (e.g. latency, execution time, clock drifting, ...)

Network behaviour

Node behaviour

Timing guarantees

Network behaviour

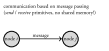
We start by assuming **bidirectional point-to-point** communication between nodes.
So network is represented as an *undirected connected graph*.

communication based on message passing
(send / receive primitives, no shared memory!)

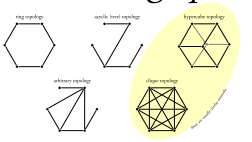
point-to-point

connected graph

Network behaviour



We start by assuming **bidirectional point-to-point** communication between nodes.
So network is represented as an *undirected connected graph*.



Links (channels) between nodes can have at least 4 possible behaviours:

- **Reliable & FIFO**:
  A message is received iff it is sent. Messages are received in the same order they are sent.
- **Reliable & non-FIFO**:
  A message is received iff it is sent. Messages can be reordered.
- **Fair-loss**:
  Messages can be lost, duplicated, reordered. If one keeps retrying, a message eventually gets through.
- **Adversarial**:
  A malicious adversary may interfere with messages (eavesdrop, modify, drop, spoof, replay).

- **Fair-loss**:
  Messages can be lo

Use of *cryptography*
(signed messages, sessions, ...)
Example: TLS

- **Adversarial**:
  A malicious advers

- **Reliable & non-F**

  A message is receiv

Use of
retry + deduplication
Example: TCP

- **Fair-loss**:

  Messages can be lo

Use of *cryptography*
(signed messages, sessions, ...)
Example: TLS

- **Reliable & FIFO**:
  A message is receiv

Use of *sequence numbers*

- **Reliable & non-F**
  A message is receiv

Use of

Node behaviour

Each node executes a *specified algorithm*,
and usually has a *unique identifier* (execution of the algorithm may depend on this).

algorithm,

Node behaviour

Each node executes a *specified algorithm*,
and usually has a *unique identifier* (execution of the algorithm may depend on this).

Failure to correctly execute the specified algorithm can result in at least 3 possible behaviours:

- **Crash-stop**:
  A node may crash at any moment. After crashing, it stops executing forever.
- **Crash-recovery**:
  A node may crash at any moment, losing its in-memory state.
  It may resume execution some time later, in which case it can recover data stored on disk.
- **Adversarial / Byzantine**:
  A node may deviate from the specified algorithm and do anything,
  including crashing or any malicious action (blocking of messages, replaying, forging, ...).

A node that is not faulty is called **correct**.

Timing guarantees

We may assume one of the following timing guarantees (*):

Synchronous **Asynchronous**

(*) in different areas of Computer Science the terms *synchronous* / *asynchronous* have different meanings.

Synchronous

- message latency no greater than a *known bound*
- nodes execute algorithm at a *known minimum speed*
- clock drift (i.e. difference between clock speeds at two nodes) within a *known range*.

Advantages:

- execution can happen in *rounds*
- mechanisms for *fault detection* can be easily implemented (e.g. time-outs)
- possibility of *synchronizing clocks* (Network Time Protocol, Precision Time Protocol)

Asynchronous

No timing guarantees at all:

- messages can be delayed arbitrarily
- nodes can pause algorithm execution
- clock speeds can diverge arbitrarily.

Advantages:

- better portability of applications.

Partially synchronous

Message latency, execution speed, and clock drift stay within limited ranges, but *bounds are not known*.

Increased difficulty w.r.t. synchronous model:

- algorithms should work correctly regardless of the actual values of the bounds.

Violations of timing guarantees in practice

Latency is predictable, most of the time. But it can occasionally increase due to:

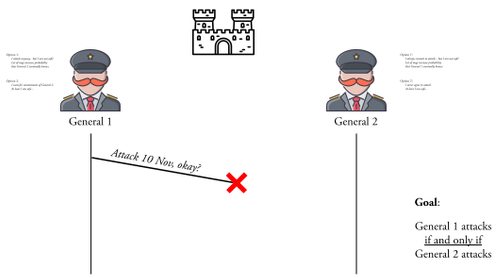
- Message loss, requiring retry
- Congestion / contention
- Network / route reconfiguration

Local code execution is comparatively fast. But pauses may occasionally happen due to:

- Operating system scheduling issues (e.g. priority inversion)
- Stop-the-world garbage collection
- Page faults, swap, thrashing, ...

Logical models

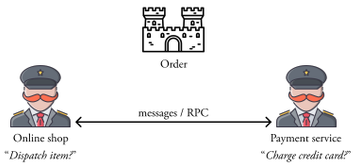
Generals problem



Goal:

General 1 attacks
if and only if
General 2 attacks

The two generals problem in practice

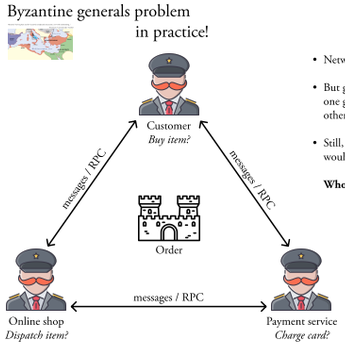


Online shop	Payment service	Outcome
Do not dispatch	*Do not charge*	*Nothing happens*
Dispatch	*Do not charge*	*Shop loses*
Do not dispatch	*Charge*	*Customer unhappy*
Dispatch	*Charge*	*Everyone happy*

Goal:

Online shop dispatches
if and only if
Payment service charges

Byzantine generals problem
in practice!



- Network is now reliable
- But generals are not:
  one general is **malicious**,
  other generals don't know who it is
- Still, honest generals
  would like to agree on a plan

Who can trust whom?

Faulty behaviours

We have seen two thought experiments:

- **Two generals problem**: a model of *faulty networks*
- **Byzantine generals problem**: a model of *faulty node behaviour*

In real systems, both nodes and networks may be faulty!

A generic (failure-oriented) **logical model** of a distributed system may capture:

- *Network behaviour* (e.g. message loss, reordering, duplication, ...)
- *Node behaviour* (e.g. crashes, malicious behaviours, ...)
- *Timing guarantees* (e.g. latency, execution time, clock drifting, ...)

Network behaviour

We start by assuming **bidirectional point-to-point** communication between nodes.
So network is represented as an *undirected connected graph*.

Links (channels) between nodes can have at least 4 possible behaviours:

- **Reliable & FIFO**:
  A message is received iff it is sent. Messages are received in the same order they are sent.
- **Reliable & non-FIFO**:
  A message is received iff it is sent. Messages can be reordered.
- **Fair-loss**:
  Messages can be lost, duplicated, reordered. If one keeps retrying, a message eventually gets through.
- **Adversarial**:
  A malicious adversary may interfere with messages (eavesdrop, modify, drop, spoof, replay).

Node behaviour

Each node executes a *specified algorithm*,
and usually has a *unique identifier* (execution of the algorithm may depend on this).

Failure to correctly execute the specified algorithm can result in at least 3 possible behaviours:

- **Crash-stop**:
  A node may crash at any moment. After crashing, it stops executing forever.
- **Crash-recovery**:
  A node may crash at any moment, losing its in-memory state.
  It may resume execution some time later, in which case it can recover data stored on disk.
- **Adversarial / Byzantine**:
  A node may deviate from the specified algorithm and do anything,
  including crashing or any malicious action (blocking of messages, replaying, forging, ...).

A node that is not faulty is called **correct**.

Timing guarantees

We may assume one of the following timing guarantees (*):

Synchronous

Asynchronous

Partially synchronous

Violations of timing
guarantees in practice

(*) in different areas of Computer Science the terms
synchronous / asynchronous have different meanings.

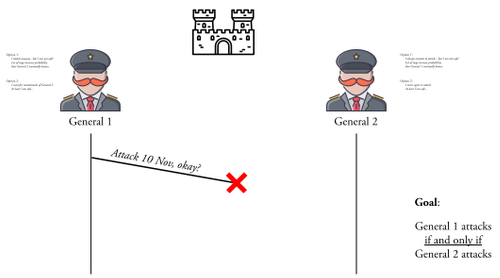
A logical model

e.g. reliable network + crash-stop + synchronous syst.
vs adversary network + Byzantine + asynchronous syst.

If your assumptions are wrong, all bets are off!

Logical models

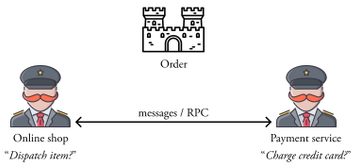
Generals problem



Goal:

General 1 attacks
if and only if
General 2 attacks

The two generals problem in practice

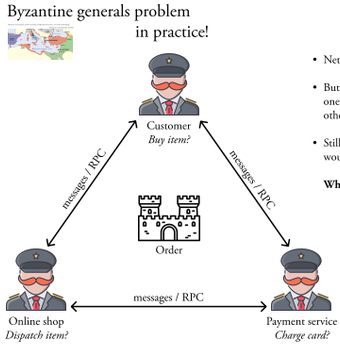


Online shop	Payment service	Outcome
Do not dispatch	*Do not charge*	*Nothing happens*
Dispatch	*Do not charge*	*Shop loses*
Do not dispatch	*Charge*	*Customer unhappy*
Dispatch	*Charge*	*Everyone happy*

Goal:

Online shop dispatches
if and only if
Payment service charges

Byzantine generals problem
in practice!



- Network is now reliable
- But generals are not:
  one general is **malicious**,
  other generals don't know who it is
- Still, honest generals
  would like to agree on a plan

Who can trust whom?

Faulty behaviours

We have seen two thought experiments:

- **Two generals problem**: a model of *faulty networks*
- **Byzantine generals problem**: a model of *faulty node behaviour*

In real systems, both nodes and networks may be faulty!

A generic (failure-oriented) **logical model** of a distributed system may capture:

- *Network behaviour* (e.g. message loss, reordering, duplication, ...)
- *Node behaviour* (e.g. crashes, malicious behaviours, ...)
- *Timing guarantees* (e.g. latency, execution time, clock drifting, ...)

Network behaviour

We start by assuming **bidirectional point-to-point** communication between nodes.
So network is represented as an *undirected connected graph*.

Links (channels) between nodes can have at least 4 possible behaviours:

- **Reliable & FIFO**:
  A message is received iff it is sent. Messages are received in the same order they are sent.
- **Reliable & non-FIFO**:
  A message is received iff it is sent. Messages can be reordered.
- **Fair-loss**:
  Messages can be lost, duplicated, reordered. If one keeps retrying, a message eventually gets through.
- **Adversarial**:
  A malicious adversary may interfere with messages (eavesdrop, modify, drop, spoof, replay).

Node behaviour

Each node executes a *specified algorithm*,
and usually has a *unique identifier* (execution of the algorithm may depend on this).

Failure to correctly execute the specified algorithm can result in at least 3 possible behaviours:

- **Crash-stop**:
  A node may crash at any moment. After crashing, it stops executing forever.
- **Crash-recovery**:
  A node may crash at any moment, losing its in-memory state.
  It may resume execution some time later, in which case it can recover data stored on disk.
- **Adversarial / Byzantine**:
  A node may deviate from the specified algorithm and do anything,
  including crashing or any malicious action (blocking of messages, replaying, forging, ...).

A node that is not faulty is called **correct**.

Timing guarantees

We may assume one of the following timing guarantees (*):

Synchronous

Asynchronous

Partially synchronous

Violations of timing
guarantees in practice

(*) in different areas of Computer Science the terms
synchronous / asynchronous have different meanings.

A logical model

e.g. reliable network + crash-stop + synchronous syst.
vs adversary network + Byzantine + asynchronous syst.

If your assumptions are wrong, all bets are off!

A good practice: try to avoid single points of failure.

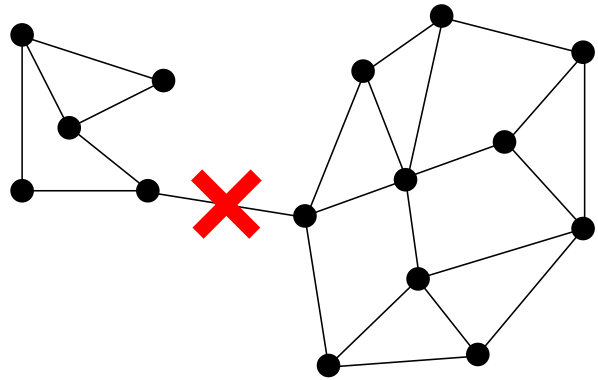
Network partitioning

Network partitioning

Network / node faults may sometimes lead to a **partitioning of the network** (a.k.a. “split brain”), where some nodes may become disconnected from other nodes, and may assume the latter have crashed.

Even if this fault were temporary, it could lead to:

- *inconsistency* (e.g. of replicated data)
- *unavailability* (e.g. of a resource or a service)



CAP Theorem

A system can be either (strongly) **consistent (C)** or **available (A)** in presence of **network partition (P)**.

consistent (C)

Every access to a resource returns the value / state
set by the "most recent" update, or possibly an error.

Or equally, at every "moment", all nodes have
the *same view* on available common resources.

available (A)

Every access to a resource returns a non-error value / state.

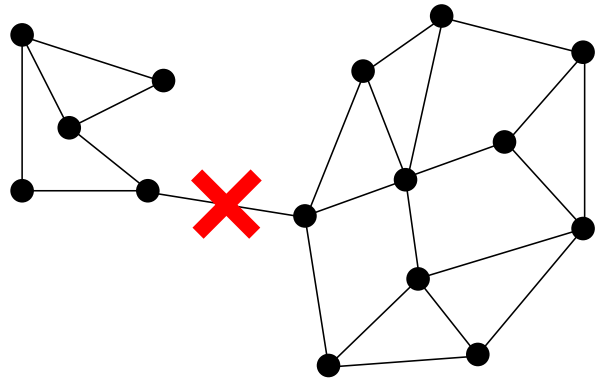
Or equally, at every moment, all nodes have access to all resources.

Network partitioning

Network / node faults may sometimes lead to a **partitioning of the network** (a.k.a. “split brain”), where some nodes may become disconnected from other nodes, and may assume the latter have crashed.

Even if this fault were temporary, it could lead to:

- *inconsistency* (e.g. of replicated data)
- *unavailability* (e.g. of a resource or a service)



CAP Theorem

A system can be either (strongly) **consistent (C)** or **available (A)** in presence of **network partition (P)**.

Every access to a resource returns the value / state set by the “most recent” update, or possibly an error.

Or equally, at every “instance”, all nodes have the *same view* on available common resources.

Every access to a resource returns a non-error value / state.

Or equally, at every moment, all nodes have access to all resources.

Consistent + Partition-tolerant

Available + Partition-tolerant

Consistent + Partition-tolerant

To avoid inconsistencies system is suspended in case of network partition.

Examples:

- ATMs
- most relational DBMSs

Available + **P**artition-tolerant

System is available at any time, but different nodes can see different information.

Examples:

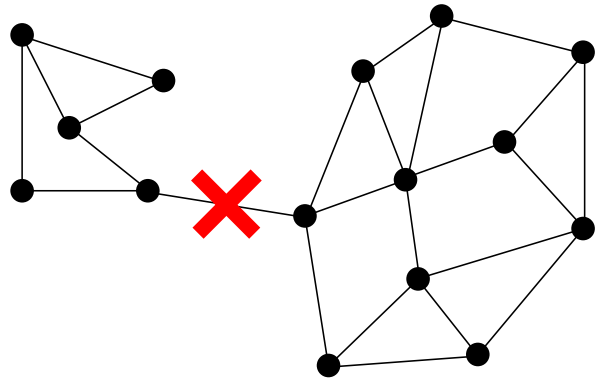
- GIT
- many "NoSQL" DBMSs

Network partitioning

Network / node faults may sometimes lead to a **partitioning of the network** (a.k.a. “split brain”), where some nodes may become disconnected from other nodes, and may assume the latter have crashed.

Even if this fault were temporary, it could lead to:

- *inconsistency* (e.g. of replicated data)
- *unavailability* (e.g. of a resource or a service)



CAP Theorem

A system can be either (strongly) **consistent (C)** or **available (A)** in presence of **network partition (P)**.

Every access to a resource returns the value / state set by the “most recent” update, or possibly an error.

Or equally, at every “moment”, all nodes have the *same view* on available common resources.

Every access to a resource returns a non-error value / state.

Or equally, at every moment, all nodes have access to all resources.

Consistent + **P**artition-tolerant

To avoid inconsistencies system is suspended in case of network partition.

Examples:
- ATMs
- most relational DBMSs

Available + **P**artition-tolerant

System is available at any time, but different nodes can see different information.

Examples:
- GIT
- many “NoSQL” DBMSs

Eventually **C**onsistent + **A**vailable + **P**artition-tolerant

Eventually **C**onsistent + **A**vailable + **P**artition-tolerant

If no new updates are made to a given resource,
then *eventually* (but no guarantees how long this might take),
from all nodes, all accesses to that resource will return the last udpated value / state.

Note: due to possible concurrent updates, this requires a mechanism for **conflict resolution**.

Examples:

- Blockchain
- Amazon Web Services

Fault tolerance

Fault: *some part* of the system is not working correctly, e.g.

- node fault (e.g. crash-stop, crash-recovery, Byzantine deviation, ...)
- network fault (e.g. dropping or significantly delaying msgs, evedropping or forgery of msgs, ...)

Failure: *system as a whole* is not functioning correctly (e.g. it is temporarily unavailable)

Fault tolerance: system as a whole continues working, despite faults (up to some maximum number of faults).

Usually requires mechanisms for *fault detection*.

fault detection.

Algorithm that detects whether a node / link is faulty.

Example: send message, await response, tag node as faulty if no reply within some timeout

Problem: cannot immediately tell difference between
crashed node, temporarily unresponsive node,
lost message, or delayed message...
(fault detection cannot be instantaneous)

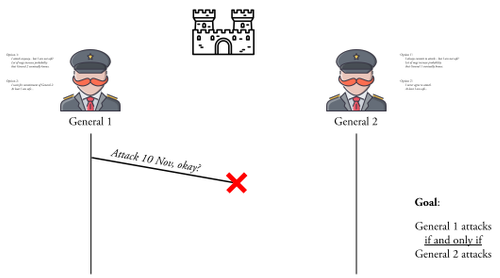
Eventually perfect fault detection

- may temporarily tag a node / link as faulty, even though node / link is correct
- may temporarily tag a node / link as correct, even though node / link is faulty
- at any moment, if no more faults / recoveries happen thereafter, then the algorithm eventually tags a node / link as faulty iff it's indeed faulty.

Logical models

[back]

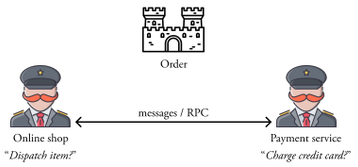
Generals problem



Goal:

General 1 attacks
if and only if
General 2 attacks

The two generals problem in practice

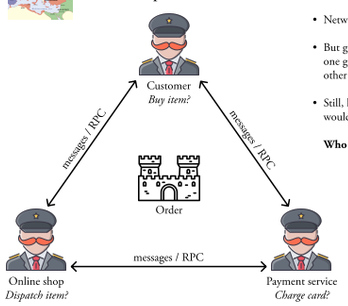


Online shop	Payment service	Outcome
Do not dispatch	*Do not charge*	*Nothing happens*
Dispatch	*Do not charge*	*Shop loses*
Do not dispatch	*Charge*	*Customer unhappy*
Dispatch	*Charge*	*Everyone happy*

Goal:

Online shop dispatches
if and only if
Payment service charges

Byzantine generals problem
in practice!



- Network is now reliable
- But generals are not:
  one general is **malicious**,
  other generals don't know who it is
- Still, honest generals
  would like to agree on a plan

Who can trust whom?

Faulty behaviours

We have seen two thought experiments:

- **Two generals problem**: a model of *faulty networks*
- **Byzantine generals problem**: a model of *faulty node behaviour*

In real systems, both nodes and networks may be faulty!

A generic (failure-oriented) **logical model** of a distributed system may capture:

- *Network behaviour* (e.g. message loss, reordering, duplication, ...)
- *Node behaviour* (e.g. crashes, malicious behaviours, ...)
- *Timing guarantees* (e.g. latency, execution time, clock drifting, ...)

Network behaviour

We start by assuming **bidirectional point-to-point** communication between nodes.
So network is represented as an *undirected connected graph*.

Links (channels) between nodes can have at least 4 possible behaviours:

- **Reliable & FIFO**:
  A message is received iff it is sent. Messages are received in the same order they are sent.
- **Reliable & non-FIFO**:
  A message is received iff it is sent. Messages can be reordered.
- **Fair-loss**:
  Messages can be lost, duplicated, reordered. If one keeps retrying, a message eventually gets through.
- **Adversarial**:
  A malicious adversary may interfere with messages (eavesdrop, modify, drop, spoof, replay).

Node behaviour

Each node executes a *specified algorithm*,
and usually has a *unique identifier* (execution of the algorithm may depend on this).

Failure to correctly execute the specified algorithm can result in at least 3 possible behaviours:

- **Crash-stop**:
  A node may crash at any moment. After crashing, it stops executing forever.
- **Crash-recovery**:
  A node may crash at any moment, losing its in-memory state.
  It may resume execution some time later, in which case it can recover data stored on disk.
- **Adversarial / Byzantine**:
  A node may deviate from the specified algorithm and do anything,
  including crashing or any malicious action (blocking of messages, replaying, forging, ...).

A node that is not faulty is called **correct**.

Timing guarantees

We may assume one of the following timing guarantees (*):

Synchronous

Asynchronous

Partially synchronous

Violations of timing
guarantees in practice

(*) in different areas of Computer Science the terms
synchronous / *asynchronous* have different meanings.

A logical model

e.g. reliable network + crash-stop + synchronous syst.
vs adversary network + Byzantine + asynchronous syst.

If your assumptions are wrong, all bets are off!

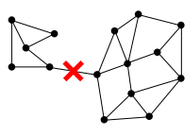
A good practice: try to avoid single points of failure.

Network partitioning

Network / node faults may sometimes lead to a **partitioning of the network** (a.k.a. "split brain"),
where some nodes may become disconnected from other nodes, and may assume the latter have crashed.

Even if this fault were temporary, it could lead to:

- *inconsistency* (e.g. of replicated data)
- *unavailability* (e.g. of a resource or a service)



CAP Theorem

A system can be either (strongly) **consistent (C)** or **available (A)** in presence of **network partition (P)**.

C$_{\text{onsistent}}$ + **P**$_{\text{artition-tolerant}}$

A$_{\text{vailable}}$ + **P**$_{\text{artition-tolerant}}$

E$_{\text{ventually}}$**C**$_{\text{onsistent}}$ + **A**$_{\text{vailable}}$ + **P**$_{\text{artition-tolerant}}$

Fault tolerance

Fault: *some part* of the system is not working correctly, e.g.
- node fault (e.g. crash-stop, crash-recovery, Byzantine deviation, ...)
- network fault (e.g. dropping or significantly delaying msgs, evedropping or forgery of msgs, ...)

Failure: *system as a whole* is not functioning correctly (e.g. it is temporarily unavailable)

Fault tolerance: system as a whole continues working, despite faults
(up to some maximum number of faults).

Usually requires mechanisms for *fault detection*.