Distributed Systems

The second half of *Concurrent and Distributed Systems*

https://www.cl.cam.ac.uk/teaching/current/ConcDisSys

Dr. Martin Kleppmann (mk428@cam)

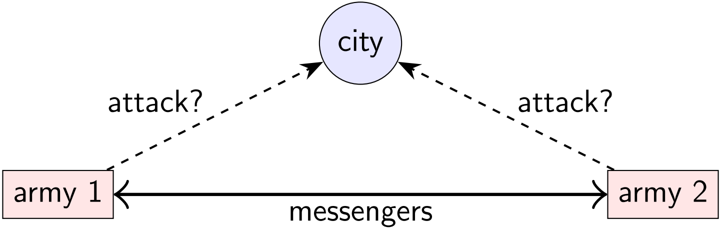
University of Cambridge

Computer Science Tripos, Part IB

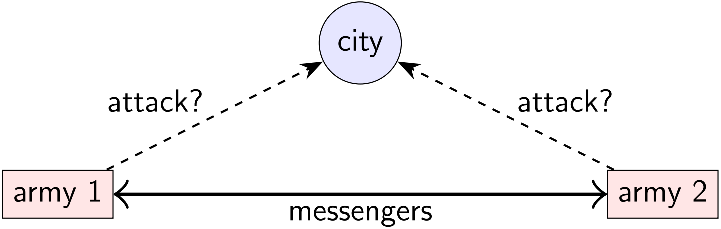
Lecture 2

Models of distributed systems

The two generals problem



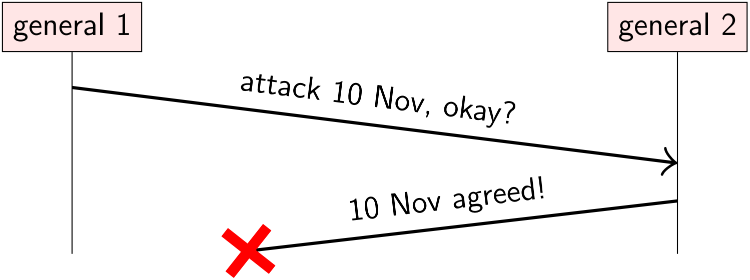
The two generals problem



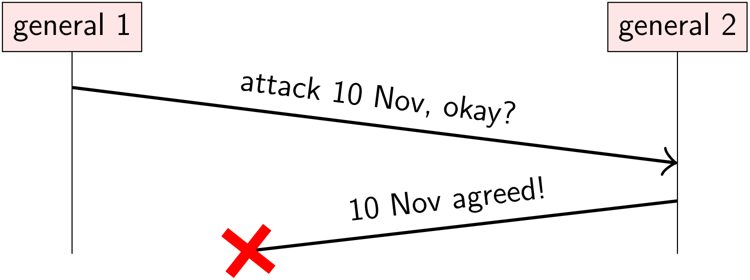
army 1	army 2	outcome
does not attack	does not attack	nothing happens
attacks	does not attack	army 1 defeated
does not attack	attacks	army 2 defeated
attacks	attacks	city captured

Desired: army 1 attacks *if and only if* army 2 attacks

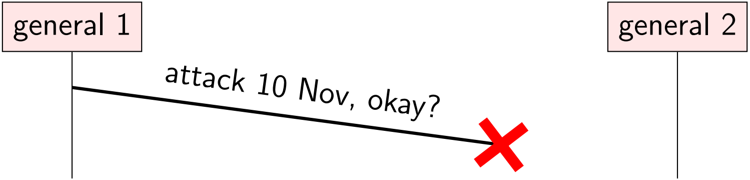
The two generals problem



The two generals problem



From general 1's point of view, this is indistinguishable from:



How should the generals decide?

1. General 1 always attacks, even if no response is received?
    - Send lots of messengers to increase probability that one will get through
    - If all are captured, general 2 does not know about the attack, so general 1 loses

How should the generals decide?

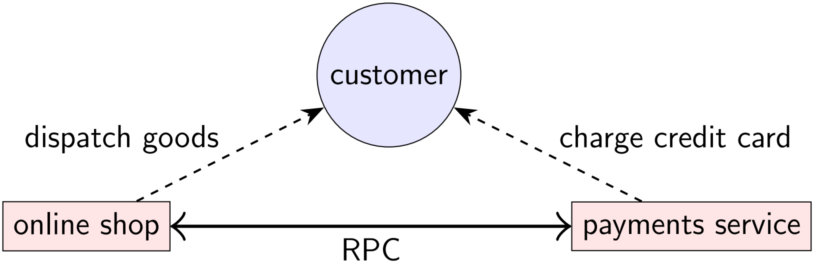
1. General 1 always attacks, even if no response is received?
   - Send lots of messengers to increase probability that one will get through
   - If all are captured, general 2 does not know about the attack, so general 1 loses
2. General 1 only attacks if positive response from general 2 is received?
   - Now general 1 is safe
   - But general 2 knows that general 1 will only attack if general 2's response gets through
   - Now general 2 is in the same situation as general 1 in option 1

How should the generals decide?

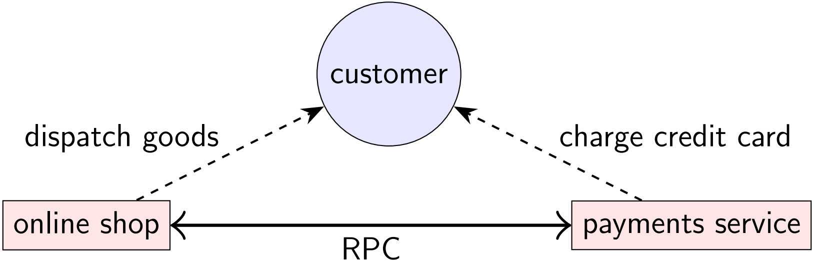
1. General 1 always attacks, even if no response is received?
   - Send lots of messengers to increase probability that one will get through
   - If all are captured, general 2 does not know about the attack, so general 1 loses
2. General 1 only attacks if positive response from general 2 is received?
   - Now general 1 is safe
   - But general 2 knows that general 1 will only attack if general 2's response gets through
   - Now general 2 is in the same situation as general 1 in option 1

No common knowledge: the only way of knowing something is to communicate it

The two generals problem applied



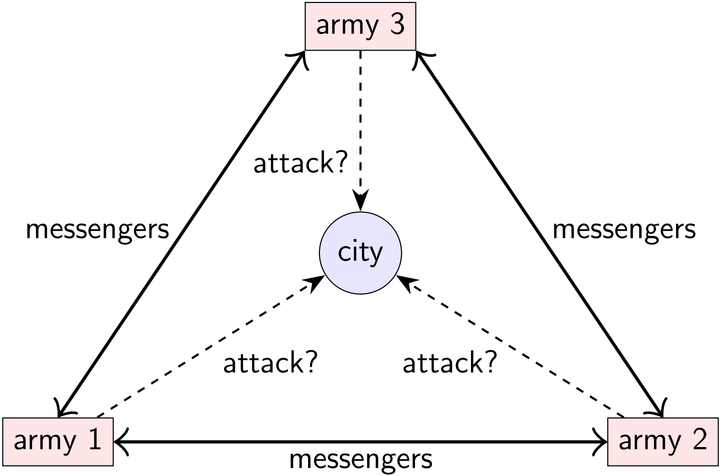
The two generals problem applied



online shop	payments service	outcome
does not dispatch	does not charge	nothing happens
dispatches	does not charge	shop loses money
does not dispatch	charges	customer complaint
dispatches	charges	everyone happy

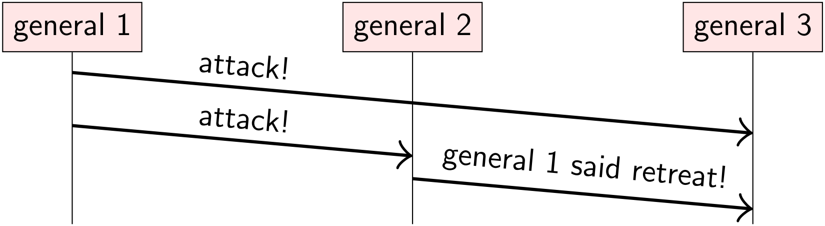
Desired: online shop dispatches *if and only if* payment made

The Byzantine generals problem

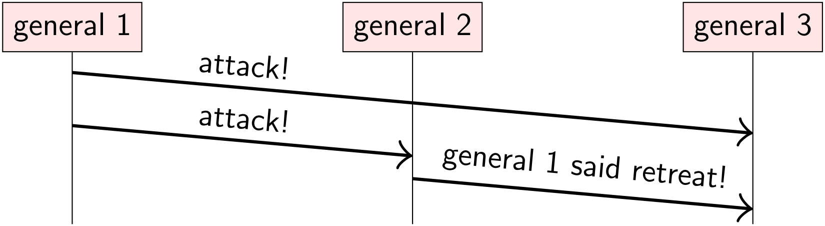


Problem: some of the generals might be traitors

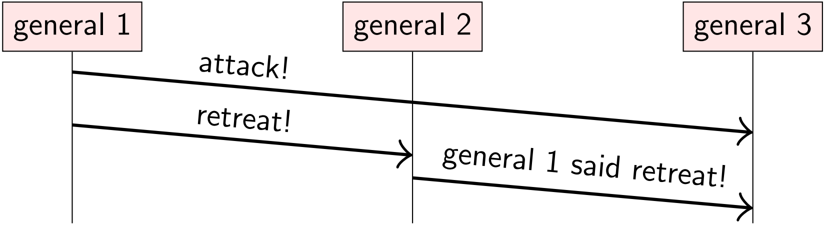
Generals that might lie



Generals that might lie



From general 3's point of view, this is indistinguishable from:



The Byzantine generals problem

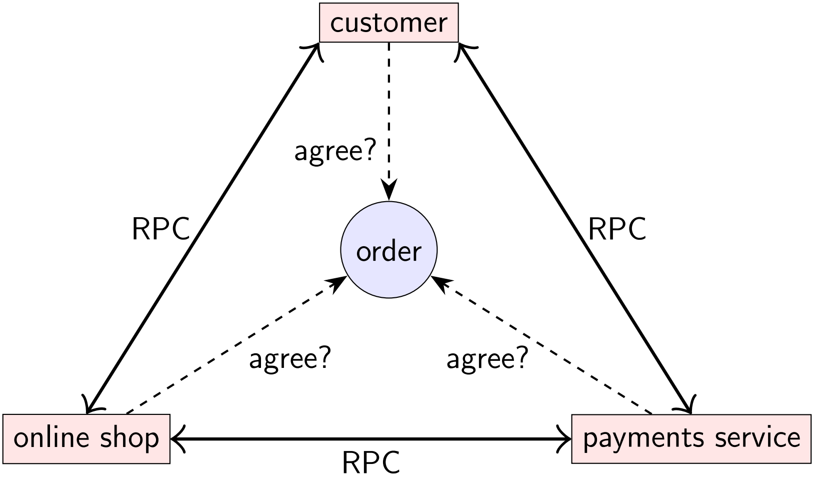
- Up to f generals might behave maliciously
- Honest generals don't know who the malicious ones are
- The malicious generals may collude
- Nevertheless, honest generals must agree on plan

The Byzantine generals problem

- Up to f generals might behave maliciously
- Honest generals don't know who the malicious ones are
- The malicious generals may collude
- Nevertheless, honest generals must agree on plan

- Theorem: need $3f + 1$ generals in total to tolerate f malicious generals (i.e. $< \frac{1}{3}$ may be malicious)
- Cryptography (digital signatures) helps – but problem remains hard

Trust relationships and malicious behaviour



Who can trust whom?

The Byzantine empire (650 CE)

Byzantium/Constantinople/Istanbul

Source: https://commons.wikimedia.org/wiki/File:Byzantiumby650AD.svg

"Byzantine" has long been used for "excessively complicated, bureaucratic, devious" (e.g. *"the Byzantine tax law"*)

System models

We have seen two thought experiments:

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

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

System models

We have seen two thought experiments:

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

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

Capture assumptions in a **system model** consisting of:

- Network behaviour (e.g. message loss)
- Node behaviour (e.g. crashes)
- Timing behaviour (e.g. latency)

Choice of models for each of these parts.

Networks are unreliable

In the sea, sharks bite fibre optic cables

https://slate.com/technology/2014/08/shark-attacks-threaten-google-s-undersea-internet-cables-video.html

On land, cows step on the cables

https://twitter.com/uhoelzle/status/1263333283107991558

System model: network behaviour

Assume bidirectional **point-to-point** communication between two nodes, with one of:

System model: network behaviour

Assume bidirectional **point-to-point** communication between two nodes, with one of:

- **Reliable** (perfect) links:
  A message is received if and only if it is sent.
  Messages may be reordered.

System model: network behaviour

Assume bidirectional **point-to-point** communication between two nodes, with one of:

- **Reliable** (perfect) links:
  A message is received if and only if it is sent.
  Messages may be reordered.
- **Fair-loss** links:
  Messages may be lost, duplicated, or reordered.
  If you keep retrying, a message eventually gets through.

System model: network behaviour

Assume bidirectional **point-to-point** communication between two nodes, with one of:

- **Reliable** (perfect) links:
  A message is received if and only if it is sent.
  Messages may be reordered.
- **Fair-loss** links:
  Messages may be lost, duplicated, or reordered.
  If you keep retrying, a message eventually gets through.
- **Arbitrary** links (active adversary):
  A malicious adversary may interfere with messages (eavesdrop, modify, drop, spoof, replay).

System model: network behaviour

Assume bidirectional **point-to-point** communication between two nodes, with one of:

- **Reliable** (perfect) links:
  A message is received if and only if it is sent.
  Messages may be reordered.
- **Fair-loss** links:
  Messages may be lost, duplicated, or reordered.
  If you keep retrying, a message eventually gets through.
- **Arbitrary** links (active adversary):
  A malicious adversary may interfere with messages (eavesdrop, modify, drop, spoof, replay).

Network partition: some links dropping/delaying all messages for extended period of time

System model: network behaviour

Assume bidirectional **point-to-point** communication between two nodes, with one of:

- **Reliable** (perfect) links:
  A message is received if and only if it is sent.
  Messages may be reordered.
- **Fair-loss** links:
  Messages may be lost, duplicated, or reordered.
  If you keep retrying, a message eventually gets through.
- **Arbitrary** links (active adversary):
  A malicious adversary may interfere with messages (eavesdrop, modify, drop, spoof, replay).

retry + dedup

Network partition: some links dropping/delaying all messages for extended period of time

System model: network behaviour

Assume bidirectional **point-to-point** communication between two nodes, with one of:

- **Reliable** (perfect) links:
  A message is received if and only if it is sent.
  Messages may be reordered.

  retry + dedup

- **Fair-loss** links:
  Messages may be lost, duplicated, or reordered.
  If you keep retrying, a message eventually gets through.

  TLS

- **Arbitrary** links (active adversary):
  A malicious adversary may interfere with messages (eavesdrop, modify, drop, spoof, replay).

Network partition: some links dropping/delaying all messages for extended period of time

System model: node behaviour

Each node executes a specified algorithm,
assuming one of the following:

- **Crash-stop** (fail-stop):
  A node is faulty if it crashes (at any moment).
  After crashing, it stops executing forever.

System model: node behaviour

Each node executes a specified algorithm,
assuming one of the following:

- **Crash-stop** (fail-stop):
  A node is faulty if it crashes (at any moment).
  After crashing, it stops executing forever.
- **Crash-recovery** (fail-recovery):
  A node may crash at any moment, losing its in-memory state. It may resume executing sometime later.

System model: node behaviour

Each node executes a specified algorithm,
assuming one of the following:

- **Crash-stop** (fail-stop):
  A node is faulty if it crashes (at any moment).
  After crashing, it stops executing forever.
- **Crash-recovery** (fail-recovery):
  A node may crash at any moment, losing its in-memory state. It may resume executing sometime later.
- **Byzantine** (fail-arbitrary):
  A node is faulty if it deviates from the algorithm.
  Faulty nodes may do anything, including crashing or malicious behaviour.

A node that is not faulty is called **“correct”**

System model: synchrony (timing) assumptions

Assume one of the following for network and nodes:

- **Synchronous**:
  Message latency no greater than a known upper bound.
  Nodes execute algorithm at a known speed.

System model: synchrony (timing) assumptions

Assume one of the following for network and nodes:

- **Synchronous**:
  Message latency no greater than a known upper bound.
  Nodes execute algorithm at a known speed.
- **Partially synchronous**:
  The system is asynchronous for some finite (but unknown) periods of time, synchronous otherwise.

System model: synchrony (timing) assumptions

Assume one of the following for network and nodes:

- **Synchronous**:
  Message latency no greater than a known upper bound.
  Nodes execute algorithm at a known speed.
- **Partially synchronous**:
  The system is asynchronous for some finite (but unknown) periods of time, synchronous otherwise.
- **Asynchronous**:
  Messages can be delayed arbitrarily.
  Nodes can pause execution arbitrarily.
  No timing guarantees at all.

Note: other parts of computer science use the terms "synchronous" and "asynchronous" differently.

Violations of synchrony in practice

Networks usually have quite predictable latency, which can occasionally increase:

- Message loss requiring retry
- Congestion/contention causing queueing
- Network/route reconfiguration

Violations of synchrony in practice

Networks usually have quite predictable latency, which can occasionally increase:

- Message loss requiring retry
- Congestion/contention causing queueing
- Network/route reconfiguration

Nodes usually execute code at a predictable speed, with occasional pauses:

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

Real-time operating systems (RTOS) provide scheduling guarantees, but most distributed systems do not use RTOS

System models summary

For each of the three parts, pick one:

- **Network:**
  reliable, fair-loss, or arbitrary
- **Nodes:**
  crash-stop, crash-recovery, or Byzantine
- **Timing:**
  synchronous, partially synchronous, or asynchronous

This is the basis for any distributed algorithm.
If your assumptions are wrong, all bets are off!

Availability

Online shop wants to sell stuff 24/7!
Service unavailability = downtime = losing money

Availability = uptime = fraction of time that a service is functioning correctly

- "Two nines" = 99% up = down 3.7 days/year
- "Three nines" = 99.9% up = down 8.8 hours/year
- "Four nines" = 99.99% up = down 53 minutes/year
- "Five nines" = 99.999% up = down 5.3 minutes/year

Availability

Online shop wants to sell stuff 24/7!
Service unavailability = downtime = losing money

Availability = uptime = fraction of time that a service is functioning correctly

- "Two nines" = 99% up = down 3.7 days/year
- "Three nines" = 99.9% up = down 8.8 hours/year
- "Four nines" = 99.99% up = down 53 minutes/year
- "Five nines" = 99.999% up = down 5.3 minutes/year

Service-Level Objective (SLO):
e.g. "99.9% of requests in a day get a response in 200 ms"

Service-Level Agreement (SLA):
contract specifying some SLO, penalties for violation

Achieving high availability: fault tolerance

Failure: system as a whole isn't working

Fault: some part of the system isn't working

- Node fault: crash (crash-stop/crash-recovery), deviating from algorithm (Byzantine)
- Network fault: dropping or significantly delaying messages

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

Single point of failure (SPOF):
node/network link whose fault leads to failure

Failure detectors

Failure detector:
algorithm that detects whether another node is faulty

Perfect failure detector:
labels a node as faulty if and only if it has crashed

Failure detectors

Failure detector:
algorithm that detects whether another node is faulty

Perfect failure detector:
labels a node as faulty if and only if it has crashed

Typical implementation for crash-stop/crash-recovery:
send message, await response, label node as crashed if no reply within some timeout

Failure detectors

Failure detector:
algorithm that detects whether another node is faulty

Perfect failure detector:
labels a node as faulty if and only if it has crashed

Typical implementation for crash-stop/crash-recovery:
send message, await response, label node as crashed if no reply within some timeout

Problem:
cannot tell the difference between crashed node, temporarily unresponsive node, lost message, and delayed message

Failure detection in partially synchronous systems

Perfect timeout-based failure detector exists only in a synchronous crash-stop system with reliable links.

Eventually perfect failure detector:

- May *temporarily* label a node as crashed, even though it is correct
- May *temporarily* label a node as correct, even though it has crashed
- But *eventually*, labels a node as crashed if and only if it has crashed

Reflects fact that detection is not instantaneous, and we may have spurious timeouts