Chapter 1
Introduction:
Basics, Models, 2 Generals

Theory of

Distributed Systems

SS 2019

Fabian Kuhn

Lecture Overview

Objectives

- Theoretical basics of distributed systems and algorithms
- Will cover a pretty diverse set of topics
- Lecture will be a mix of the previous Distributed Systems lecture and of the previous Network Algorithms lecture

General topics

- Coordination and agreement
- Faults and asynchrony
- Global states and time
- Distributed lower bound / impossibility proofs
- Distributed network / graph algorithms
- Massively parallel graph computations

Lecture Organization

Lecture and Exercises (101-01-009/013)

- Lecture: Monday 14:15 – 16:00
- Exercises: Wednesday 14:15 – 16:00

Format of the lecture

- Weekly lecture and exercise sheets
- We will discuss some exercises together and will give you some more that you can try to solve by yourself
- Doing the exercises is not mandatory, it is however highly recommended
- Material covered in the exercises is also part of the oral exam!
- Try to keep the lecture interactive
  - Please ask questions!

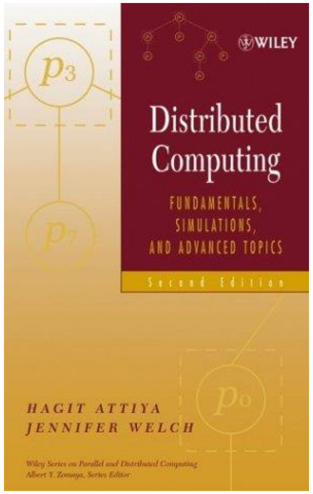
Web Page

http://ac.informatik.uni-freiburg.de

→ Teaching → SS 2019 → (Theory of) Distributed Systems

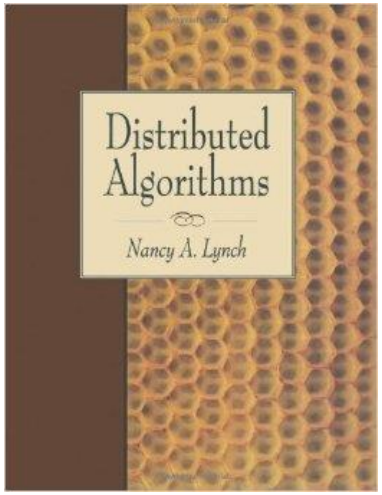
- We will publish all important information there!
  - Slides, lecture notes, exercises, recordings, links to further literature, …
- Check the web page regularly!
- Recordings and/or lecture notes will be put online
  - Sometimes possibly with some delay…
- Some information will be password protected
  - password will be sent by email to all students registered for the course

Text Books


WILEY
Distributed Computing
FUNDAMENTALS, SIMULATIONS, AND ADVANCED TOPICS
Second Edition
HAGIT ATTIYA
JENNIFER WELCH
Wiley Series on Parallel and Distributed Computing
Albert Y. Zomaya, Series Editor

Distributed Computing
Fundamentals, Simulations, and Advanced Topics
(second edition)

Hagit Attiya and Jennifer Welch
John Wiley and Sons, Inc., 2004



Distributed Algorithms

Nancy Lynch
Morgan Kaufmann Publishers, Inc., 1996

Additional Literature

- **Lecture notes by James Aspnes (Yale U.):**

  Theory of Distributed Systems 2019

  http://cs.yale.edu/homes/aspnes/classes/465/notes.pdf

  - based on book by [Attiya, Welch 04]

- **Lecture material by Nancy Lynch (MIT):**

  Distributed Algorithms 2015

  https://learning-modules.mit.edu/class/index.html?uuid=/course/6/fa15/6.852#info

  - Based on book by [Lynch 98]

- **Lecture material by Roger Wattenhofer (ETH Zurich):**

  Principles of Distributed Computing

  https://disco.ethz.ch/courses/podc_allstars/

  - Basis of the old Network Algorithms lecture

- Potential additional material will be published on the webpage.

What is a Distributed System?

A distributed system is a collection of individual computing devices that can communicate with each other.

…

Each processor in a distributed system generally has its semiindependent agenda, but for various reasons, including sharing of resources, availability, and fault tolerance, processors need to coordinate their actions.

[Attiya, Welch 2004]

Why are Distributed Systems Important?

Distributed systems are everywhere!

- The Internet
- WWW
- Local area networks, corporate networks, ...
- Parallel architectures, multi-core computers
- Cell phones
- Internet applications
- Peer-to-peer networks
- Data centers
- ...

Why are Distributed Systems Important?

Distributed systems allow to

- share data between different places
- handle much larger amounts of data
- parallelize computations across many machines
- build systems that span large distances
- build communication infrastructures

and also to

- build robust and fault-tolerant systems

Why are Distributed Systems Different?

In distributed systems, we need to deal with many aspects and challenges besides the ones in non-distributed systems.

Some challenges in distributed systems:

- How to organize a distributed system
  - how to share computation / data, communication infrastructure, ...
- There is often no global time
- Coordination of multiple (potentially heterogeneous) nodes
- Coordination in networks of arbitrary (unknown) topologies
- Agreement on steps to perform
- All of this in the presence of asynchrony (unpredictable delays), message losses, and faulty, lazy, malicious, or selfish nodes

Why Theory?

For distributed systems, we don't have the kind of tools for managing complexity like in standard sequential programming!

Main reason: a lot of inherent **nondeterminism**

- unpredictable delays, failures, actions, concurrency, ...
- no node has a global view
- leads to a lot of **uncertainty**!

It is much harder to get distributed systems right

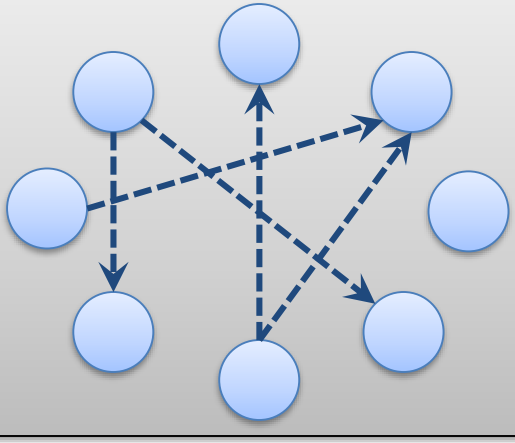
- Important to have theoretical tools to argue about correctness
- Correctness may be theoretical, but an incorrect system has practical impact!
- Easier to go from theory to practice than vice versa ...

Distributed System Models

Two basic abstract models for studying distributed systems...

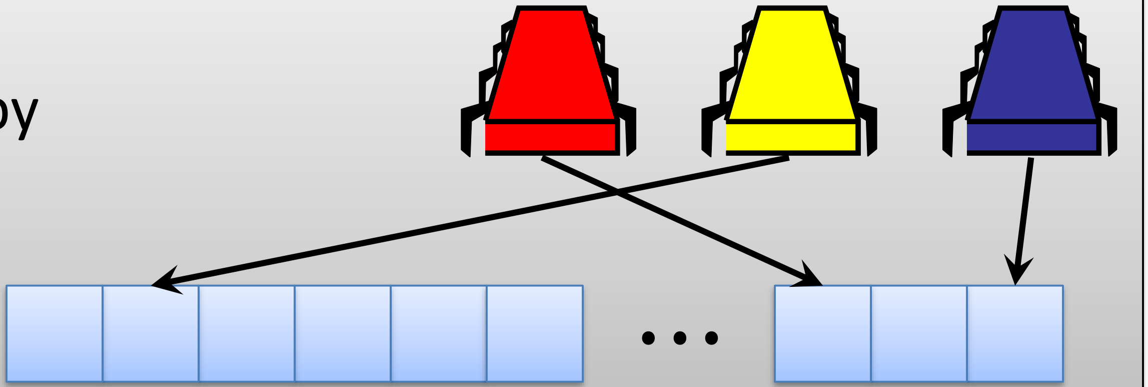
Message Passing:

- Nodes/processes interact by exchanging messages
- Fully connected topology or arbitrary network



Shared Memory:

- Processes interact by reading/writing from/to common global memory



Distributed System Models

Message Passing

- Used to model large (decentralized) systems and networks
- Except for small-scale systems, real systems are implemented based on exchanging messages
- Certainly the right model for large systems that use a large number of machines, but also for many other practical systems

Shared Memory

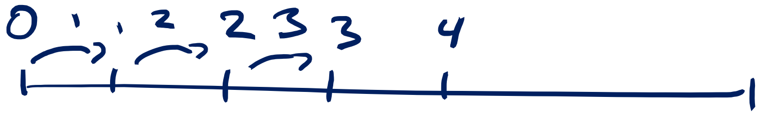
- Classic model to study many standard coordination problems
- Models multi-core processors and also multi-threaded programs on a single machine
- Most convenient abstraction for programming

Distributed System Models

Message Passing vs. Shared Memory

- Generally, the two models can simulate each other
  - One can implement the functionality of a shared memory system based on exchanging messages
  - One can implement the functionality of a message passing system based on using a shared memory
- Most things we discuss hold for both models
- We will see both models and we will switch back and forth between the models (as convenient)
  - We will mostly consider message passing algorithms

Synchrony



Synchronous systems:

- System runs in synchronous time steps (usually called **rounds**)
  - Discrete time $0, 1, 2, 3, 4, \ldots$
  - Round r takes place between time $r - 1$ and time r

Synchronous message passing:

- **Round r:**
  At time $r - 1$, each process sends out messages (or a single msg.)
  Messages are delivered and processed at time r

Synchronous shared memory:

- In each round (at each time step), every process can access one memory cell

Synchrony

Asynchronous systems:

- Process speeds and message delays are finite but otherwise completely unpredictable
- Assumption: process speeds / message delays are determined in a worst-case way by an adversarial scheduler

Asynchronous message passing:

- Messages are always delivered (in failure-free executions)
- Message delays are arbitrary (chosen by an adversary)

Asynchronous shared memory:

- All processes eventually do their next steps (if failure-free)
- Process speeds are arbitrary (chosen by an adversary)

Synchrony

There are modeling assumptions between completely synchronous and completely asynchronous systems.

- **Bounded message delays / process speeds:**
  Nodes can measure time differences and there is a (known) upper bound T on message delays / time to perform 1 step.
  - Model is **equivalent to the synchronous model**
  - 1 round $= T$ time units

- **Partial synchrony:**
  There is an upper bound on message delays / process speeds
  - Variant 1: upper bound is not known to the nodes / processes
  - Variant 2: upper bound only starts to hold at some unknown time

Failures

Crash Failure:

- A node / process stops working at some point in the execution
- Can be in the middle of a round (in synchronous systems)
  - some of the messages might already be transmitted...

Byzantine Failure:

- A node / process (starts) behaving in a completely arbitrary way
- Different Byzantine nodes might collude

Omission Failure:

- Node / process / communication link stops working temporarily
- E.g., some messages get lost

Resilience:

- Number of failing nodes / processes tolerated

Correctness of Distributed Systems

When dealing with distributed systems and protocols, there are different kinds correctness properties.

The three most important ones are...

Safety: Nothing bad ever happens

Liveness: Something good eventually happens

Fairness: Something good eventually happens to everyone

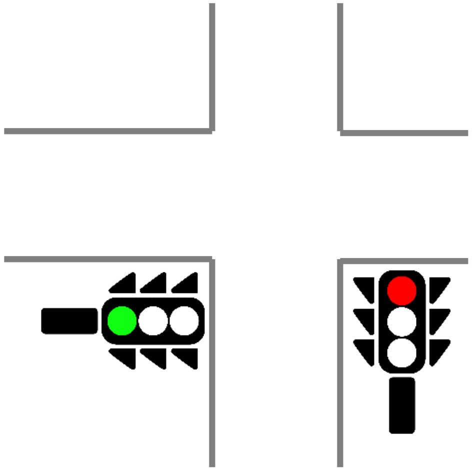
Safety

Nothing bad ever happens.

Equivalent: There are no bad reachable states in the system

Example:

- At each point in time, at most one of the two traffic lights is green.



Proving safety:

- Safety is often proved using invariants
- Every possible state transition keeps a safe system safe

Liveness

Something good eventually happens.

Example:

- My email is eventually either delivered or returned to me.

Remark:

- Not a property of a system state but of system executions
- Property must start holding at some finite time

Proving liveness:

- Proofs usually depend on other more basic liveness properties, e.g., all messages in the system are eventually delivered

Fairness

Something good eventually happens to everybody.

- Strong kind of liveness property that avoids starvation

Starvation: Some node / process cannot make progress

Example 1: System that provide food to people

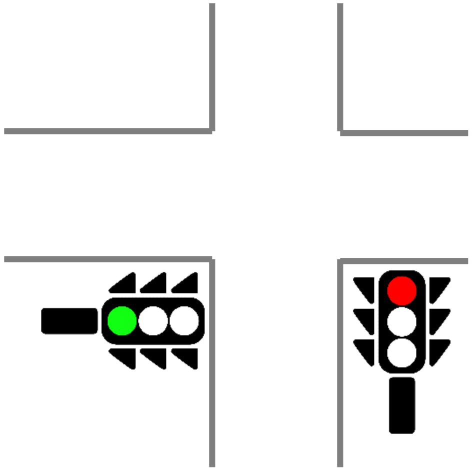
- Liveness properties:
  - Somebody gets food
  - System provides enough food for everybody

Example 2: Mutual Exclusion (exclusive access to some resource)

- Liveness properties:
  - some process can access the resource
  - the resource can be accessed infinitely often

Safety, Liveness and Fairness

Traffic Light Example



Safety: At most one of the two lights is green at each point in time.

Liveness: There is a green light infinitely often

Fairness: Both lights are green infinitely often

Message Passing : More Formally

General remark: We'll try to keep the formalism as low as possible, however some formalism is needed to argue about correctness.

- For detailed models: [Attiya,Welch 2004], [Lynch 1996]

Basic System Model:

1. System consists of n (deterministic) nodes/processes $v_1, \dots, v_n$ and of pairwise communication channels
   - implicit assumption that nodes are numbered $1, \dots, n$, n is known
   - sometimes, we want to relax this condition

   n known, but nodes are labeled with unique IDs

   upper on n known, n compl. unknown (uniform alg.)

2. At each time, each node v_i has some internal state Q_i

3. System is event-based: states change based on discrete events

Event-Based Model

Internal State of a Node:

- Inputs, local variables, possibly some local clocks
- History of the whole sequence of observed events

Types of Events:

- **Send Event:** Some node v_i puts a message on the communication channel to node v_i
- **Receive Event:** Node v_j receives a message
  - must be preceded by a corresponding send event
- **Timing Event:** Event triggered at a node by some local clock

Remarks:

- Events might trigger local computations which might trigger other events

Schedules and Executions

Configuration C: Set (vector) of all n node states (at a given time)

- configuration = system state

Execution Fragment:

Sequence of alternating configurations and events

- Example: $\underline{C_0}, \underline{\phi_1}, \underline{C_1}, \underline{\phi_2}, \underline{C_2}, \phi_3, \ldots$
  - C_i are configurations, ϕ_i are events
- Each triple $\underline{C_{i-1}, \phi_i}, C_i$ needs to be consistent with the transition rules for event ϕ_i
  - e.g., rcv. event $\underline{\phi_i}$ only affects the state of the node that received the msg.

Execution: execution fragment that starts with initial config. C_0

Schedule: execution without the configurations, but including inputs

(the sequence of events of an execution & the inputs)

Message Passing Model: Remarks

Local State:

- State of a node v_i does not include the states of messages sent by v_i (v_i doesn’t know if the message has arrived / been lost)

Adversary:

- Within the timing guarantees of the model (synchrony assumptions), execution/schedule is determined in a worst-case way (by an adversary)

Deterministic nodes:

- In the basic model, we assume that nodes are deterministic
- In some cases this will be relaxed and we consider nodes that can flip coins (randomized algorithms)
- Model details / adversary more tricky

Local Schedules

A node v's state is determined by v's inputs and observable events.

Schedule Restriction

- Given a schedule S, we define the **restriction $S|i$** as the subsequence of S consisting v_i's inputs and of of all events happening at node v_i

Example:

- 3 nodes v_1, v_2, v_3, send events s_{ij} , receive events r_{ji}
- Schedule $S = s_{13}, s_{23}, s_{31}, r_{13}, s_{32}, r_{31}, r_{23}, s_{13}, s_{21}, r_{31}, r_{12}, r_{32}$

$S|1 =$ $s_{13}, r_{13}, s_{13}, r_{12}$

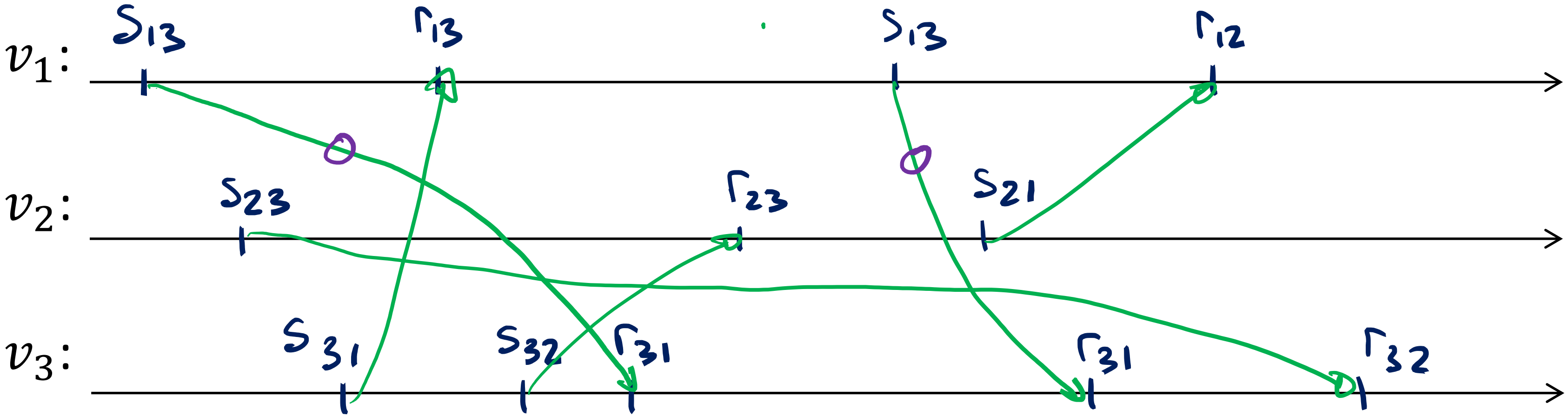
$S|2 =$ s_{23}, r_{23}, s_{21}

$S|3 =$ $s_{31}, s_{32}, r_{31}, r_{31}, r_{32}$

Graphical Representation of Executions

Schedule $S = \underline{s_{13}, s_{23}, s_{31}, r_{13}, s_{32}, r_{31}, r_{23}, s_{13}, s_{21}, r_{31}, r_{12}, r_{32}}$

Graphical representation of schedule / execution



Indistinguishability

Theorem (indistinguishability):
If for two schedules $\underline{S}$ and $\underline{S'}$ and for a node $\underline{v_i}$ with the same inputs in S and S', we have $\underline{S|i} = \underline{S'|i}$, if v_i takes the next action, it performs the same action in both schedules S and S'.

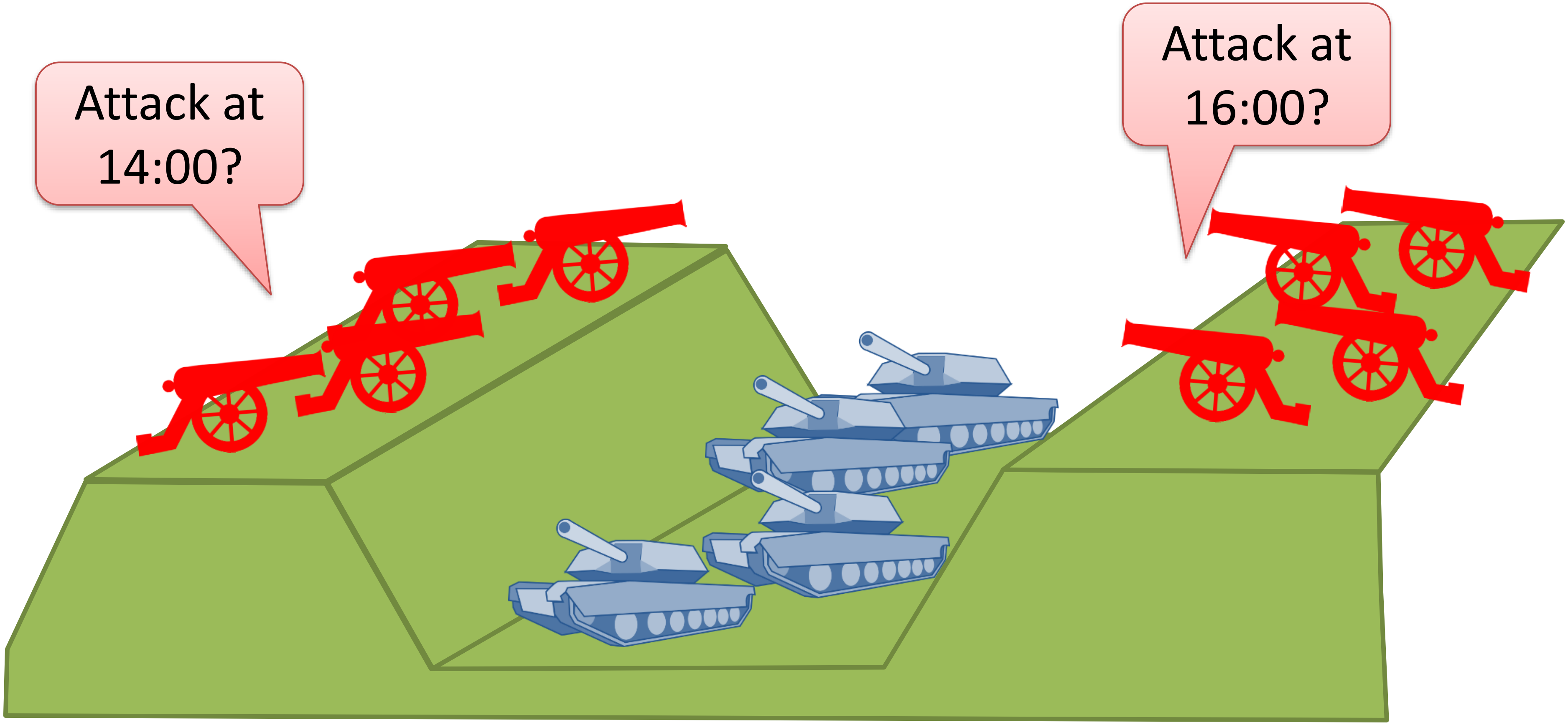
Proof:

- State of a node v_i only depends on inputs and on $\underline{S|i}$
- For deterministic nodes, the next action only depends on the current state.

Lower Bounds / Impossibility Proofs:

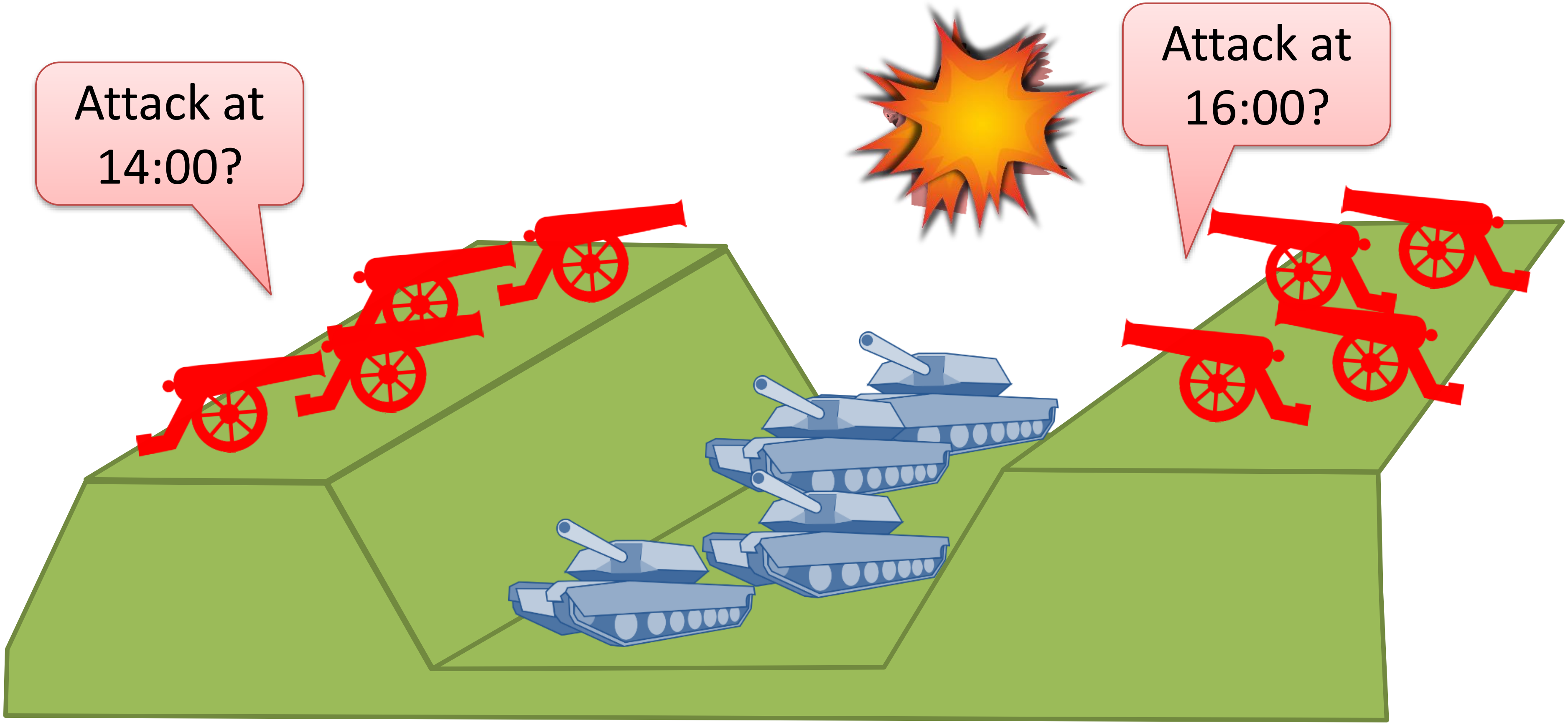
- Most lower bounds and impossibility proofs for distributed systems are based on indistinguishability arguments.

The Two Generals Problem



- To win, the two red armies need attack together
- They need to agree on a time to attack the blue army

The Two Generals Problem



- Communication across the valley only by carrier pigeons
- Problem: pigeons might not make it

The Two Generals Problem

Problem is relevant in the real world...

- Alice and Bob plan to go out on Saturday evening
- They need to agree on:
  - when and where to me~~a~~et
  - who makes the dinner reservation
  - ...
- They can only communicate by an unreliable messaging service

- Nodes in a network need to agree on
  - who's the leader for some computation
  - which of two / several conflicting data accesses to perform
  - whether to commit a distributed database transaction
  - ...

Two Generals More Formally

Model: two deterministic nodes, synchronous communication, unreliable messages (messages can be lost)

Input: each node starts with one of two possible inputs 0 or 1

- say input encodes time to attack

Output: Each node needs to decide either 0 or 1

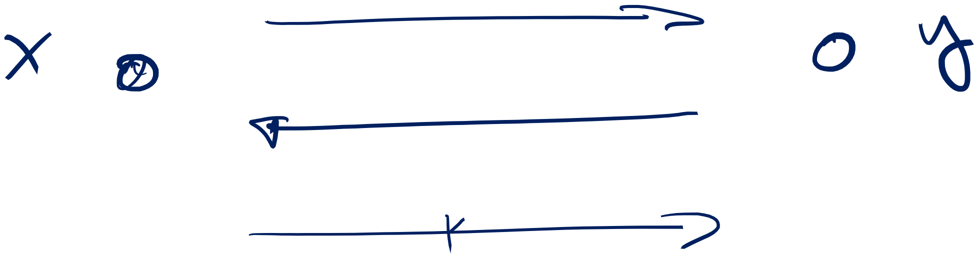
Agreement: Both nodes must output the same decision (0 or 1)

Validity: If both nodes have the same input $x \in \{0,1\}$ and no messages are lost, both nodes output x.

- If nodes start with different inputs or one or more messages are lost, nodes can output 0 or 1 as long as they agree.

Termination: Both nodes terminate in a bounded # of rounds.

Solving the Two Generals Problem?



Two Generals: Impossibility

Indistinguishability Proof:

- Execution E is indistinguishable from execution E' for some node v if v sees the same things in both executions.
  - same inputs and messages (schedule)
- If E is indistinguishable from E' for v, then v does the same thing in both executions.
  - We abuse notation and denote this by $E|v = E'|v$

Similarity:

- Consider all possible executions $E_1, E_2, \ldots$
- Call E_i and E_j **similar** if $E_i|v = E_j|v$ for some node v

$$E_i \sim_v E_j \iff E_i|v = E_j|v$$

Two Generals: Impossibility

Consider a chain $E_0, E_1, E_2, \dots, E_k$ of executions such that for all $i \in \{1, \dots, k\}$, E_{i-1} and E_i are similar.

- $\forall i \in \{1, \dots, k\} : E_{i-1} \sim_v E_i$ for some node v

$$\forall i: \; E_{i-1} \sim_v E_i \quad (E_{i-1}|v = E_i|v)$$

→ v does the same thing in E_{i-1} & E_i

→ v outputs the same decision

agreement: all nodes output the same value in E_{i-1} & E_i

$$E_0 \sim E_1 \sim E_2 \sim \dots \sim E_k$$

$$x \qquad x \qquad x \qquad\qquad x$$

Two Generals: Impossibility

Proof Idea:

- Assume there is a T-round protocol
  - Then, nodes can always decide after exactly T rounds
- Construct sequence of executions $E_0, E_1, \dots, E_k$ s.t.
  - For all $i \in \{1, \dots, k\}$ $E_{i-1} \sim_v E_i$ for some node $v \in \{v_1, v_2\}$
  - In E_0 output needs to be 0 and in E_k output needs to be 1

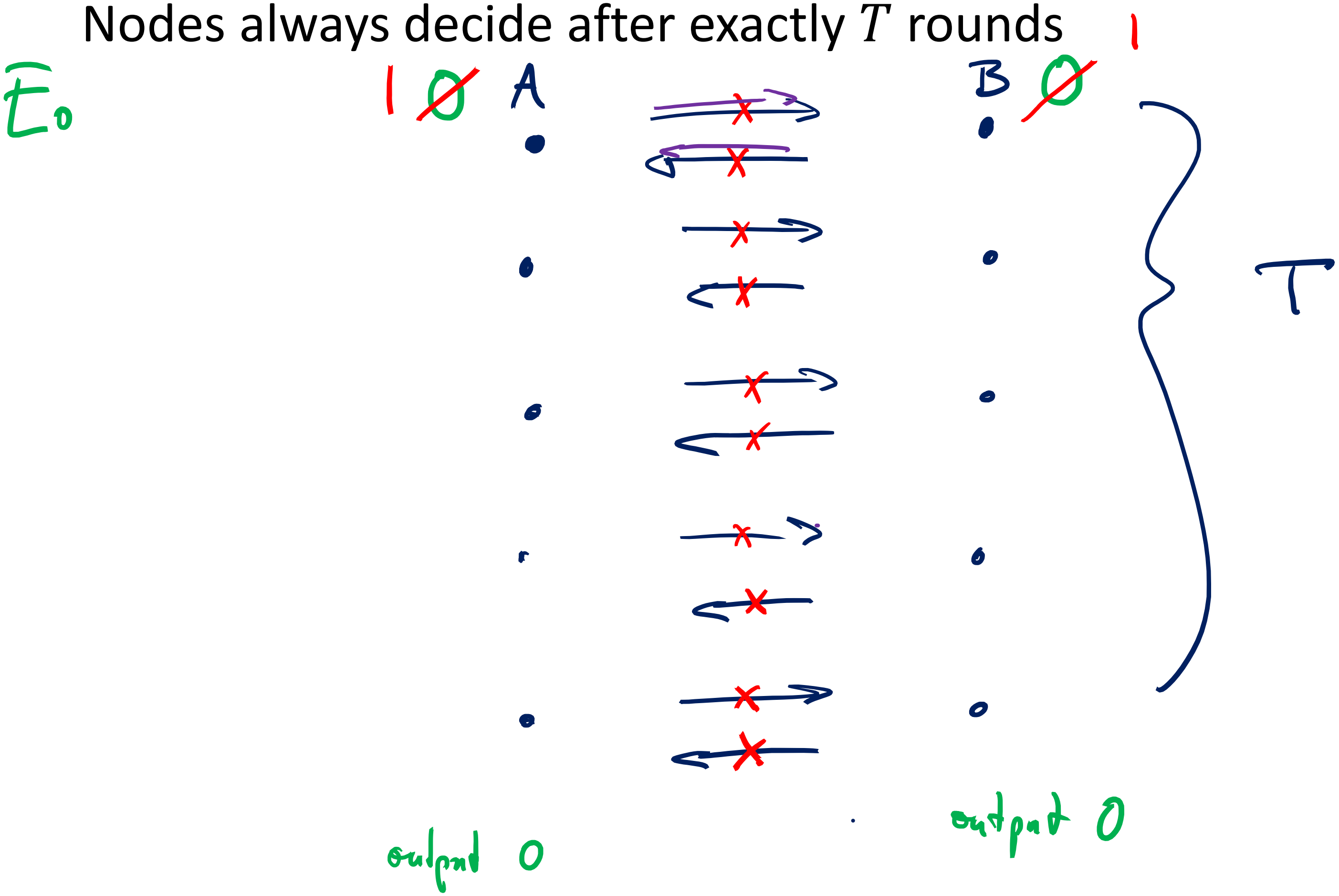
Execution E_0 : both inputs are 0, no messages are lost — both output 0

Execution E_k : both inputs are 1, no messages are lost — " " 1

Validity

Two Generals: Impossibility

Nodes always decide after exactly T rounds



Two Generals: Impossibility

Nodes always decide after exactly T rounds

Execution $\boldsymbol{E_0}$: both inputs are 0, no messages are lost

Execution $\boldsymbol{E_1}$: one of the messages in round T is lost

Execution $\boldsymbol{E_i}$: last message M is delivered in round t

Execution $\boldsymbol{E_{i+1}}$: drop message M

Execution $\boldsymbol{E_{2T}}$: both inputs are 0, no messages are delivered

- All nodes output 0 (because of similarity chain)

Two Generals: Impossibility

Execution $\boldsymbol{E_{2T}}$: both inputs are 0, no messages are delivered

- All nodes output 0 (because of similarity chain)

Execution $\boldsymbol{E_{2T+1}}$: input of v_1 is 0, input of v_2 is 1, no msg. delivered

Execution $\boldsymbol{E_{2T+2}}$: input of both nodes are 1, no msg. delivered

Execution $\boldsymbol{E_{4T+2}}$: input of both nodes are 1 and no msg. are lost

- from E_{2T+2} to E_{4T+2} deliver messages one by one
- same chain as from E_0 to E_{2T}, but in opposite direction
- **In $\boldsymbol{E_{4T+2}}$, all nodes must output 1 $\Longrightarrow$ contradiction!**

Two Generals Impossibility: Summary

- We start with an execution in which both nodes have input 0 and no messages are lost $\Longrightarrow$ both nodes must decide 0.
- We prune messages one by one to get a sequence of executions s.t. consecutive executions are similar.
- From an execution with no messages delivered and both inputs 0, we can get to an execution with no messages delivered and both inputs 1 (in two steps).
- By adding back messages one-by-one, we get to an execution in which both nodes have input 1 and no messages are lost
$\Longrightarrow$ both nodes must decide 1 $\Longrightarrow$ contradiction!

- Not hard to generalize to an arbitrary number $n \geq 2$ of nodes
- Upper bound on number of rounds not necessary
  - as long as nodes need to decide in finite time

Two Generals: Randomized Algorithm

- The two generals problem can be solved if
  - we allow (one of) the two generals to flip coins
  - we are satisfied if agreement is only achieved with probability $1-\varepsilon$ (for ε small enough)

- But first, we look at a simple algorithm:

The Level Algorithm (Overview):

- Both nodes compute a level
- At the end, the two levels differ by at most one
- The levels essentially measure the number of successful back and forth transmissions