IN.5022 — Concurrent and Distributed Computing

Foundations

Prof. P. Felber

pascal.felber@unine.ch

Agenda

- What is a distributed system?
- Architectural models
- Fundamental models

What is a distributed system?

A collection of independent computers that appears to its users as a ***single coherent system***

Provides the means for performance, scalability, dependability, etc.

What is a distributed system?

A collection of applications communicating by message passing in order to solve ***a common task***

Independent computers cooperate together

What is a distributed system?

"A distributed system is one in which
the ***failure*** of a computer you didn't even know existed
can render your own computer unusable"

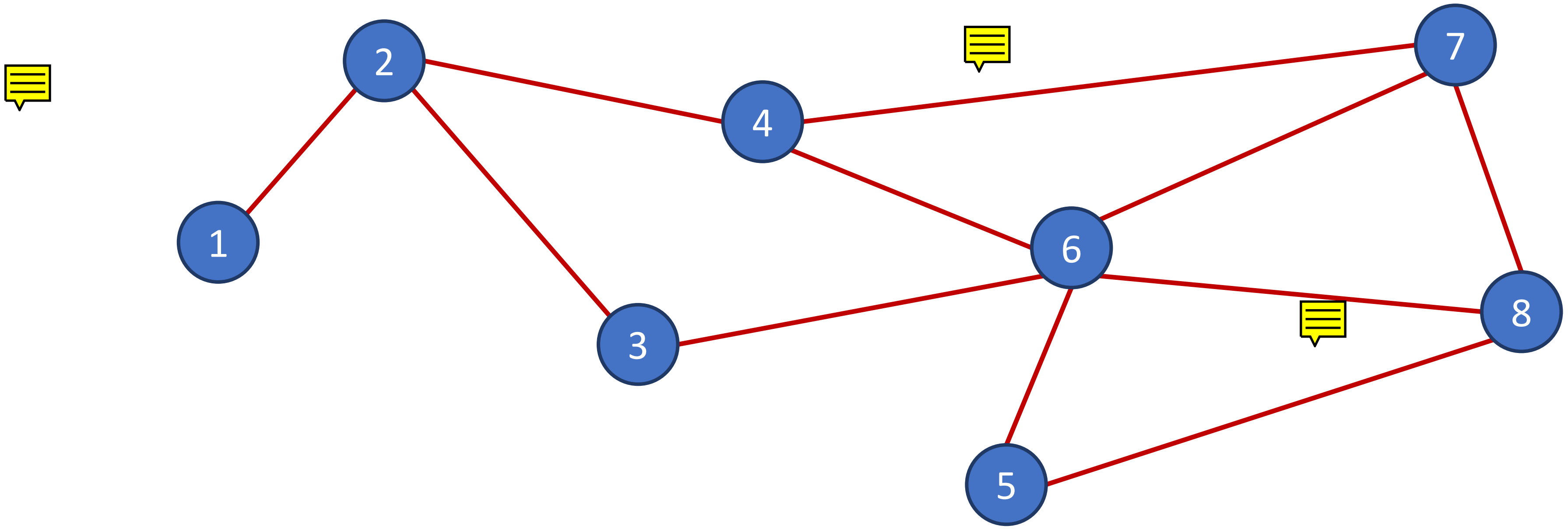
Leslie Lamport

Introduces special problems regarding
correctness, complexity, failures…

Fault tolerance: the ability of a system to provide useful service (possibly degraded in functionality and/or performance), despite the fact that some of its components malfunction

What is a distributed system?

- Abstract view: a ***network*** of ***processes***
  - *Nodes* are processes, *edges* are communication channels

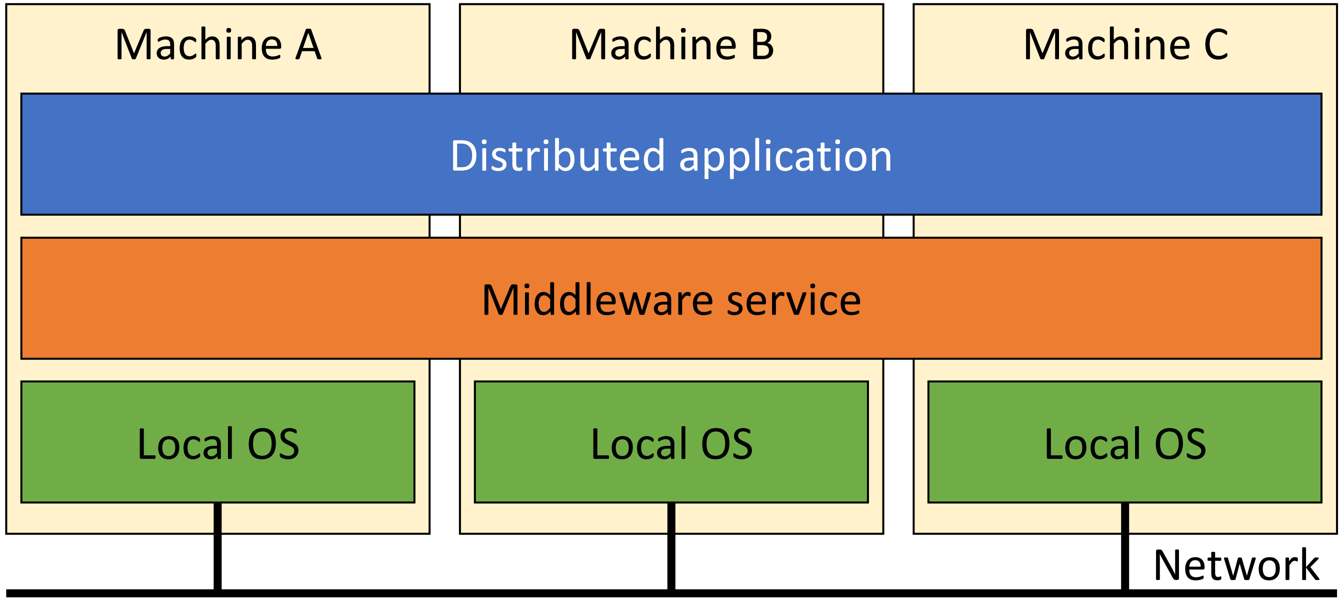


Example of distributed systems

- Internet, intranets, WWW
- Telecommunication networks
- Airline reservation systems
- Aircraft control systems
- Electronic banking (interbank networks)
- P2P, sensor networks
- Grid computing (LHC, SETI, etc.)
- Social networks

Service layers in a distributed system

- A distributed system organized as middleware
  - Middleware layer extends over multiple machines
  - Middleware masks heterogeneity and provides **transparency**

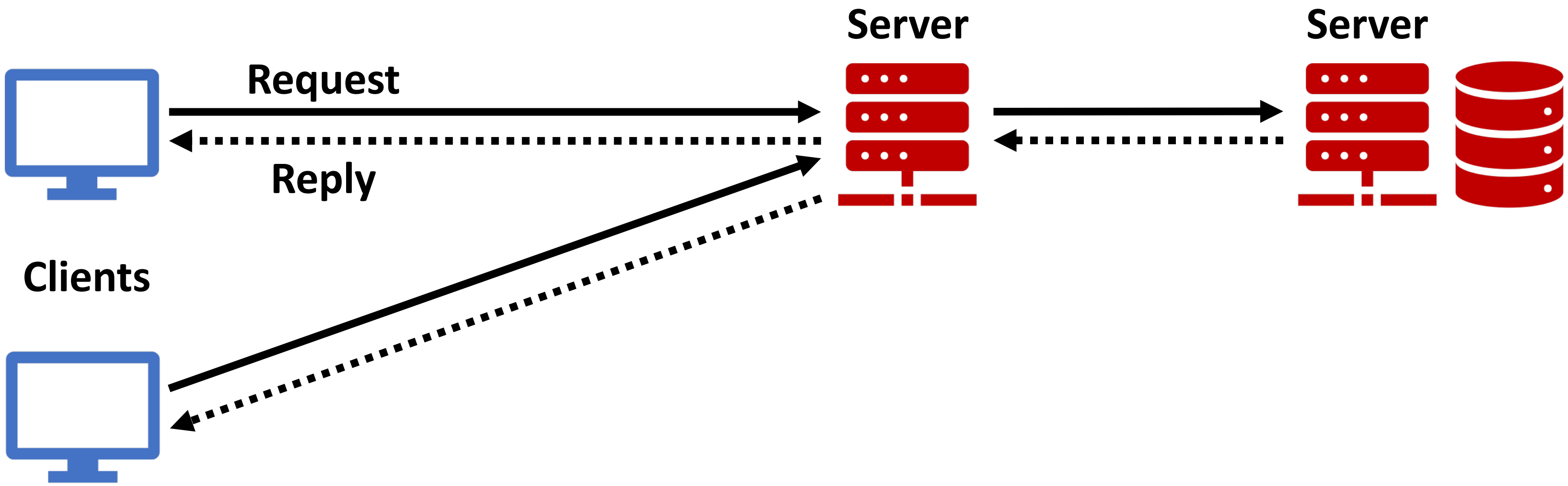


Hiding complexity in a distributed system

- A distributed system will shield the programmer from some complex problems by "transparently" handling them
  - Access transparency: hide distribution (e.g., RMI)
  - Location transparency: hide resource location (e.g., URIs)
  - Concurrency transparency: hide concurrent operation on shared resources (e.g., transactions)
  - Replication transparency: hide service or resource redundancy (e.g., CDNs)
  - Failure transparency: hide failures of hardware or software components (e.g., transparent application failover)
  - And more...

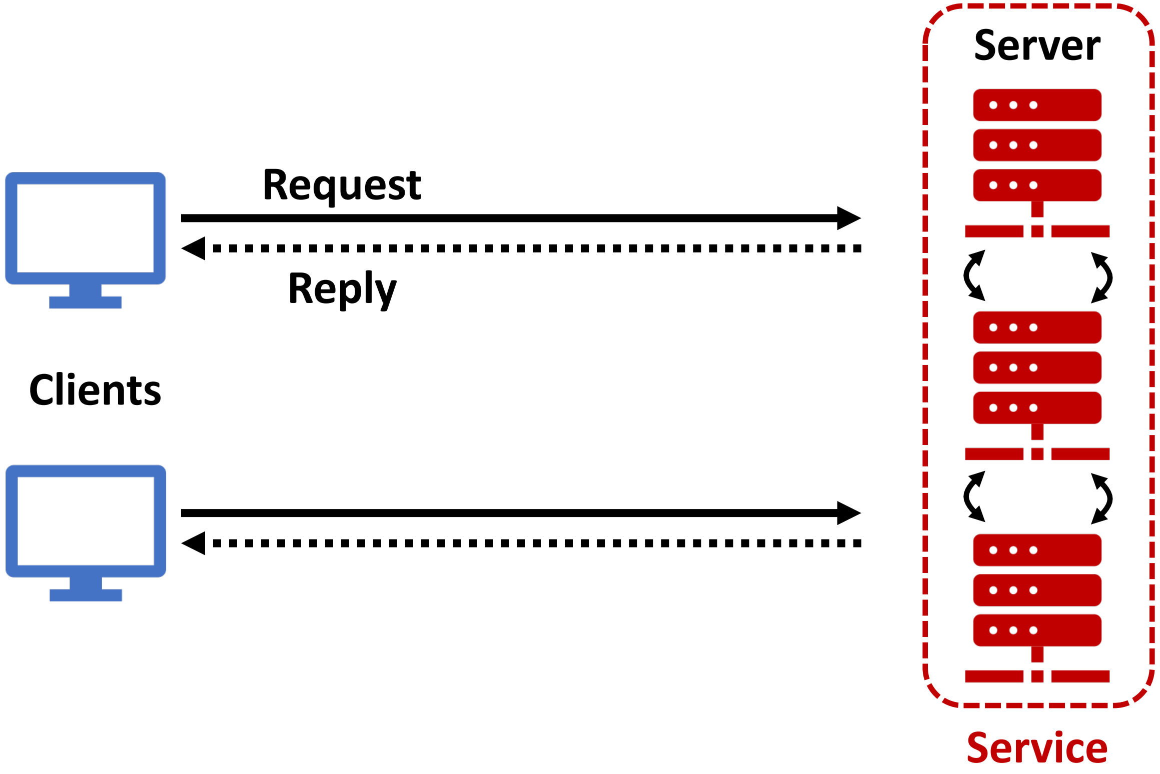
Client/server architectures

- Clients invoke individual servers
- Servers may be clients of other servers
  - E.g., browser client of a Web server, in turn client of a file server, a database or a DNS



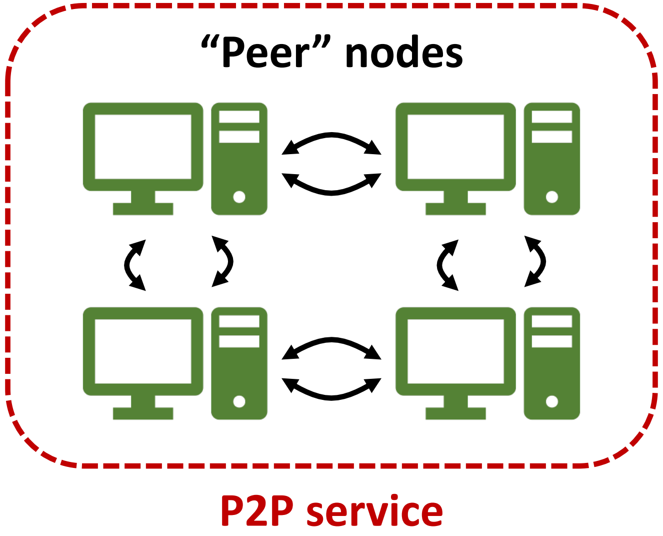
Client/server architectures

- A service can be provided by multiple servers
  - E.g., Google, Amazon



Peer architectures

- Multiple processes communicate with each other as peers
  - E.g., replicated service, file sharing networks, distributed whiteboard, internet routers



Why distribute?

- Centralized architectures do not scale well
  - Centralized services (e.g., a single server for all users)
  - Centralized data (e.g., a single on-line telephone book)
  - Centralized algorithms (e.g., routing using complete information)

- Distribution enables scaling
  - Break components and spread parts across a distributed system
  - Use of asynchronous communication
  - Replicate critical components for high availability
  - Cache data (data replication)

Why distribute?

- The **optimistic** view
  - Concurrency ⇒ speed (parallel processing, load balancing)
  - Partial failures ⇒ high availability
- The **pessimistic** view
  - Concurrency ⇒ incorrectness (interleavings)
  - Partial failures ⇒ incorrectness (inconsistent state)

Some challenges of distributed computing

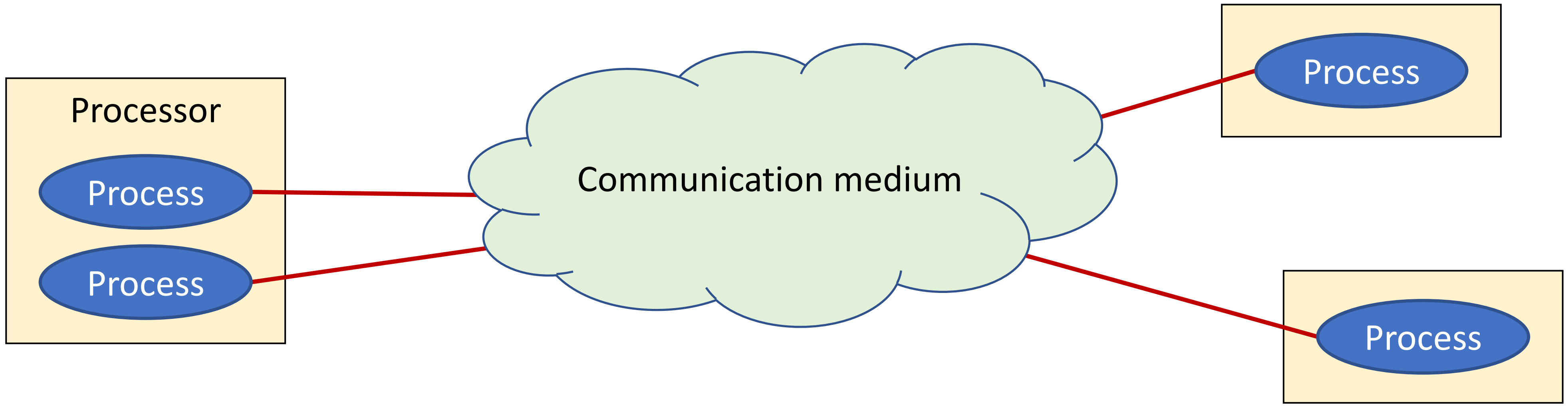
- Knowledge of a process
  - Local (identity, state, neighbours) vs. global (whole system)
- Network topology
  - Sparse topologies (e.g., ring, tree) vs. fully connected
- Degree of synchronization
  - Clock drift, message delays, processor speed
- Failure
  - Crash, omission, arbitrary
- Scalability
  - Performance not impaired by size, e.g., $O(\log N)$ vs. $O(N)$

Common subproblems

- Solving these challenges boils down to solving subproblems
  - Many applications revolve around a few common subproblems
  - Solving them helps having a good handle on system design
- Some classical examples
  - Leader election: elect one among a set of processes
  - Mutual exclusion: limit access to shared resources
  - Time synchronization: synchronize local clocks
  - Global state: collect all local states at a given time
  - Multicast/broadcast: send message to several processes
  - Replica management: keep state of replicas synchronized

Distributed system models

- Processes communicating with one another



- What problems can one solve in a distributed system?
  - It depends on the **system model**: communication, timeliness, failure behaviour, etc.

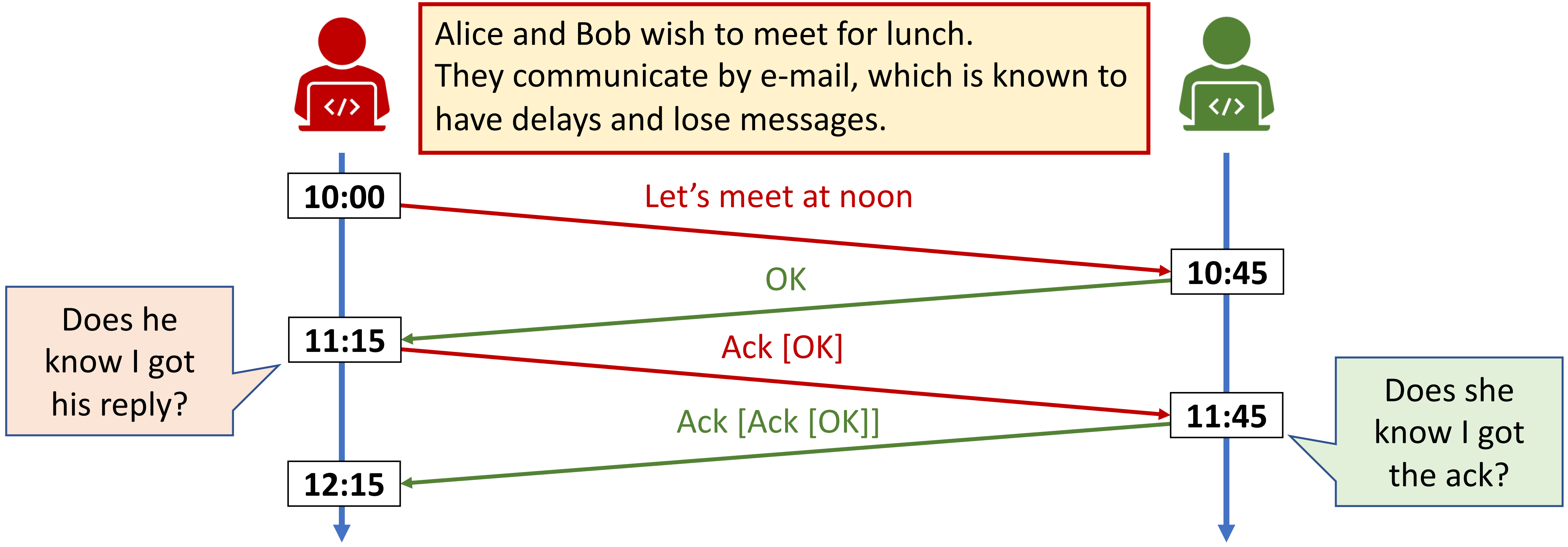
Solvability of distributed problems

- Reliable broadcast
  - Ensure that a message sent to a group of processes is received by all or by none
- Atomic commit
  - Ensure that processes reach a common decision on whether to commit or abort a transaction
- Mutual exclusion
  - Ensure that only one process executes in a “critical section” at a time

“Solvability” and algorithms depends on the system model

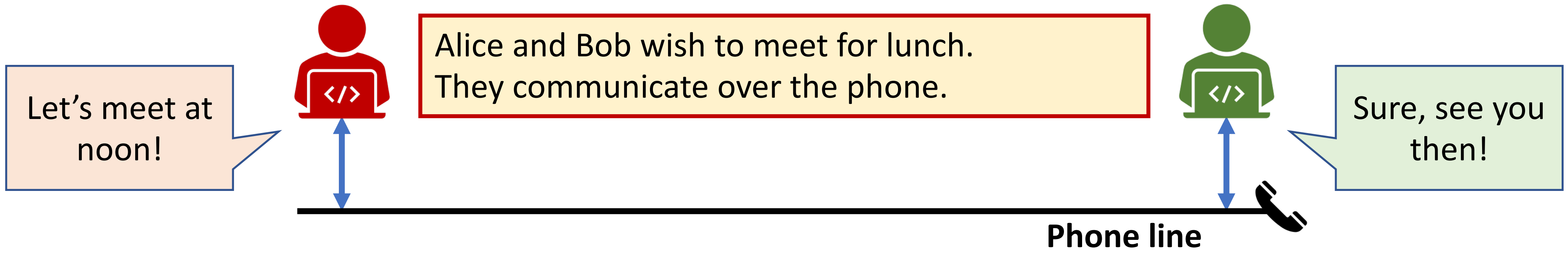
Example

Alice and Bob wish to meet for lunch.
They communicate by e-mail, which is known to have delays and lose messages.



System model: unreliable message transmission
no bounds on transmission time
no commonly accessible medium (e.g., whiteboard)

Example



System model: one party hears what the other says within a bounded delay
or
the existence of a problem is known within a bounded delay

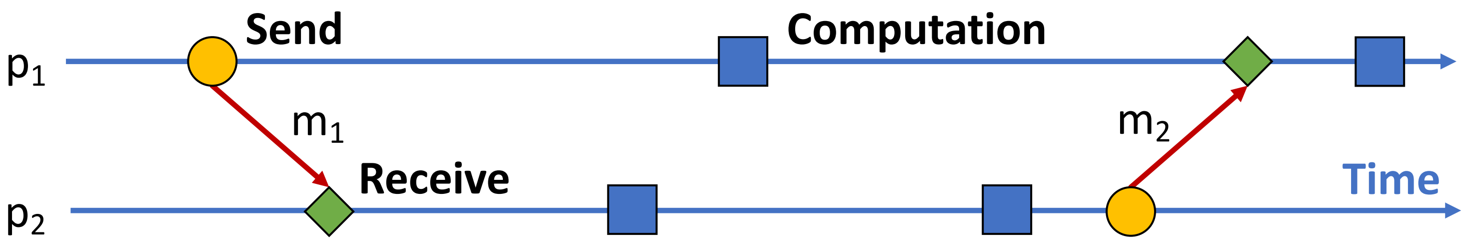
Some definitions

- A distributed system is made of a **finite set of processes** (generally denoted Π or P)
  - Each process models a **sequential** program
  - Processes are denoted by $p_1, \ldots, p_N$ or p, q, r
  - Processes **communicate** with each other by exchanging messages

Some definitions

- A process executes one step every tick of its local clock
  - A local computation (local event)
  - A message exchange (global event)
    - Sending a message to one or several processes
    - Receiving a message from one process
- A **history** of process p_i is an ordered series of events

$$h_i = \langle e_i^1, e_i^2, e_i^3, \ldots \rangle$$



Approach to distributed problems

- **Specifications**

  What is the problem?

- **Assumptions**

  What is the system model?
  What is the power of the adversary?

- **Algorithms**

  How do we solve the problem?
  How do we prove that the algorithm is correct?
  At which cost?

Specifications

- A specification describes the problem in terms of safety and liveness properties
  - **Safety:** nothing bad ever happens
  - **Liveness:** something good eventually happens

- Any specification can be expressed in terms of safety and liveness properties

Specifications

- Example of safety and liveness properties

“Tell the truth!”

Safety: *“You shall not lie!”*

Liveness: *“You have to say something”*

Assumptions: system models

- But what is a model?
  - A collection of attributes and a set of rules that govern how these attributes interact

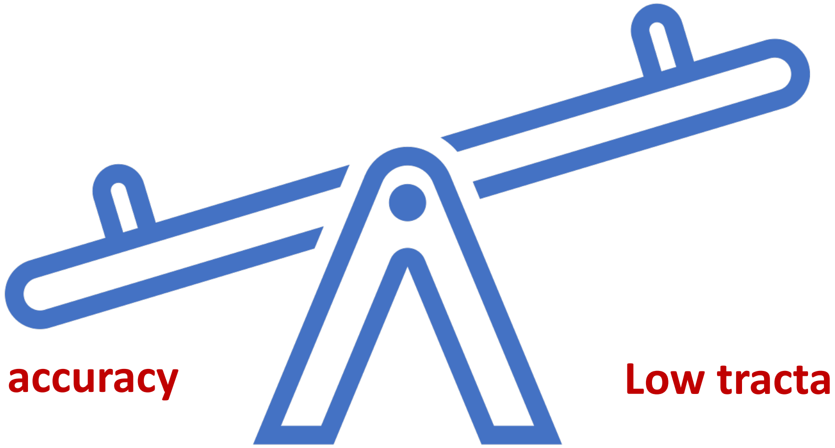
- Can a model be wrong?

"A theory has only the alternative of being right or wrong.
A model has a third possibility: it may be right, but irrelevant."

Manfred Eigen

Good models

- Accurate models
  - Yield truth about the object of interest
- Tractable models
  - Analyzing them is actually possible



Good models

- What to expect from a model?
  - Feasibility

    What classes of problems can be solved?
    (in a given model)
  - Cost

    How expensive is the solution?
    (for solvable problems)

An example: coordination

- A coordination problem
  - Processes p and q communicate by sending and receiving messages on a bidirectional channel
  - Neither process can fail, but the channel may lose messages
  - Processes can execute one of two actions
  - Devise a protocol in which both processes take the same action, and neither takes both actions

An example: coordination

- There is no solution to the problem!
  (in the given model)
- Proof (by contradiction)
  - Any protocol executes in rounds of message exchanges: first (say) p sends a message to q, then q sends a message to p, and so on
  - Let Φ be the protocol that solves the problem using the fewest rounds
  - Assume w.l.o.g. that the last message is sent by p, and let it be m

An example: coordination

- Proof (cont’d)
  - Observation #1: the action taken by p cannot depend on m, because its receipt could never be learned by p (it is the last message)
  - Observation #2: the action taken by q cannot depend on m, because q must make the same choice of action even if m is lost (due to a channel failure)

An example: coordination

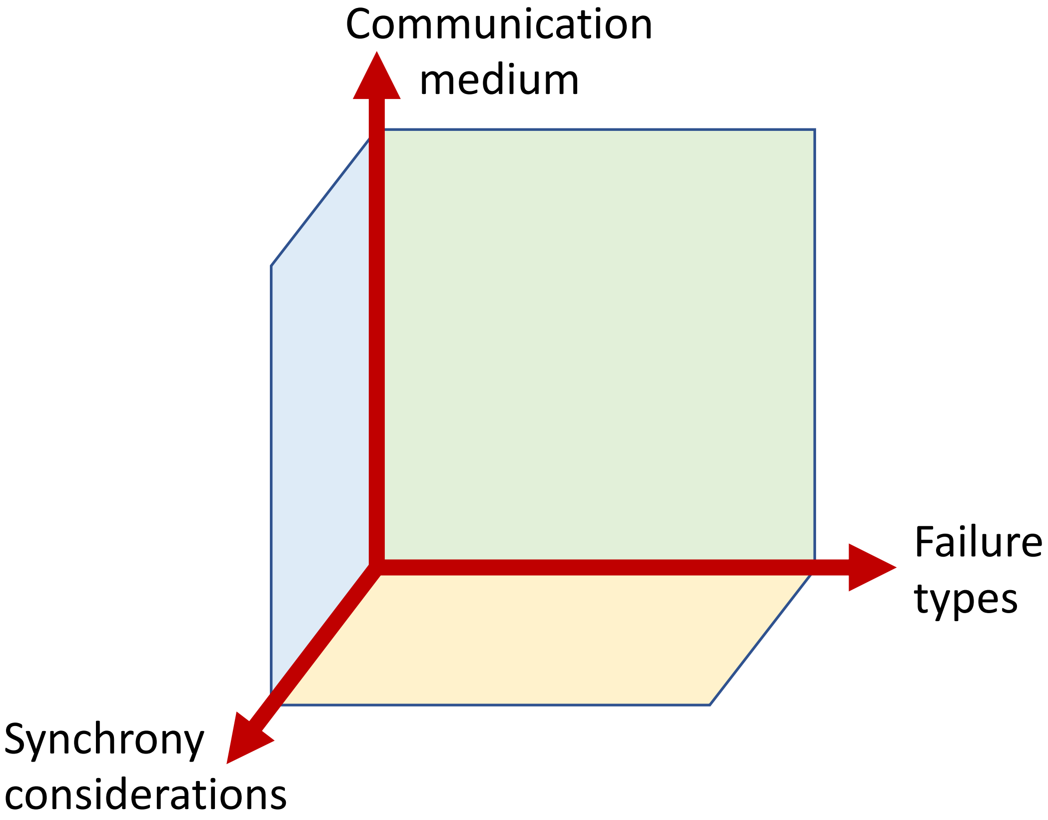
- Proof (cont'd)
  - Since the action chosen by p and q does not depend on m, it follows that m is not needed and so we can construct a new protocol in which one fewer message is sent…
    …a contradiction!

Lessons learned

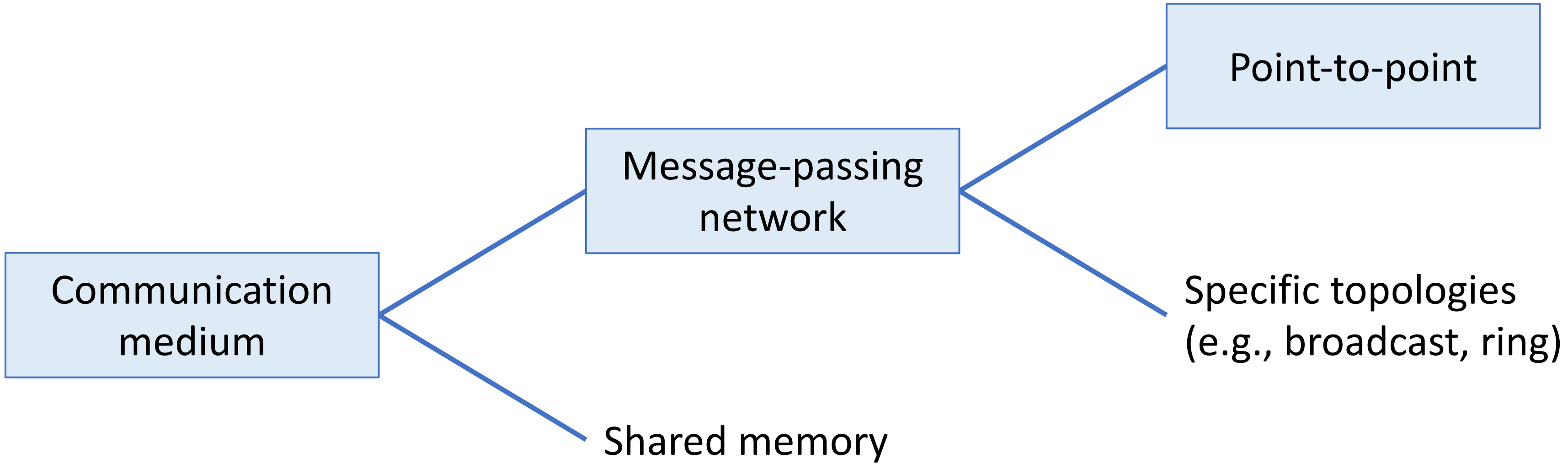
- What have we learned?
  - All protocols between two processes in this model are equivalent to a series of message exchanges
  - Actions taken by a process depend only on the sequence of messages it has received

Assumptions: system models

- 3 dimensions to consider

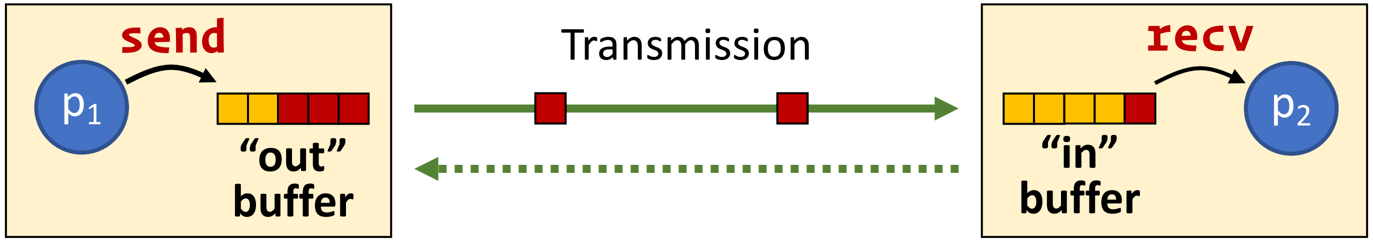


Communication medium



Communication medium

- Point-to-point networks
  - Processes connected by a link communicate via **`send`**/**`recv`**
  - Links may be uni- or bi-directional
  - Non-blocking send primitive necessary for fault-tolerance
  - Often assume fully-connected network (links between each pair of processes)
  - Links are not necessarily direct physical connections



Communication medium

- **Fair-lossy** (unreliable) point-to-point links (channels)
  - **Fair-loss**

    If a message m is sent infinitely often by p to q,
    and neither p nor q crashes,
    then m is delivered infinitely often to q

  - **Finite duplication**

    If a message m is sent a finite number of times by p to q,
    then it is delivered a finite number of times to q

  - **No creation**

    No message is delivered unless it was sent

Communication medium

- **Reliable** point-to-point links (channels)
  - **Validity**

    If p sends m to q and q does not fail,
    then q eventually receives m from p

  - **No duplication**

    Process q receives messages m from p at most once

  - **No creation**

    Process q receives messages m from p
    only if p has previously sent m to q

Failure types

- Link liveness failure: **message loss**

  …message sent from p to correct process q never received by q…

  - A link that violates (satisfies) specification is faulty (correct)

- Process liveness failure: **crash**

  …process stops taking steps…

  - A process that violates (satisfies) specification is faulty (correct)

  Crash-stop model: a crashed process never recovers (it stops taking steps forever)

Failure types

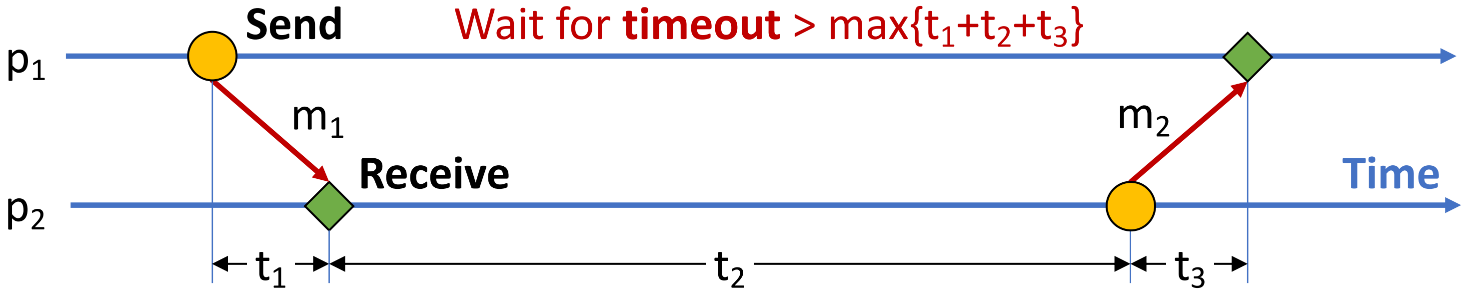
- Arbitrary failures (**Byzantine**)
  - Process/channel may send/transmit arbitrary messages at arbitrary times, or commit omissions
  - Process may take an incorrect step
- Performance failures
  - Process exceeds the bounds on the interval between two steps
  - Message transmission takes longer than the stated bound
- And more...

Synchrony considerations

- Synchronous network model
  - Known upper bound on the time required for a process to execute a local step
  - Known upper bound on message transmission delay
  - Can assume that process have synchronized (within known bounds) physical clocks

Synchrony considerations

- Consequences of synchronous model
  - Can use timeouts to detect process / link failures



- Can organize computation in round
  - Send messages to a set of processes Π
  - Receive replies of that round from all processes in Π (failures are detected using timeouts)
  - Change state

Synchrony considerations

- Asynchronous network model
  - No bound on the time required for a process to execute a local step (however, this time is finite)
  - No bound on the message transmission delay
  - Cannot assume the existence of perfectly or approximately synchronized physical clocks
- The most general model
  - An algorithm designed for asynchronous systems also works in synchronous systems

Synchrony considerations

- Unfortunately, some very basic computational problems cannot be solved in *asynchronous systems*...

  in a fault-tolerant manner

  and

  with a deterministic algorithm

- Thus, for certain problems we have to resort to...

  synchronous (or partially synchronous) systems

  or

  randomized algorithm

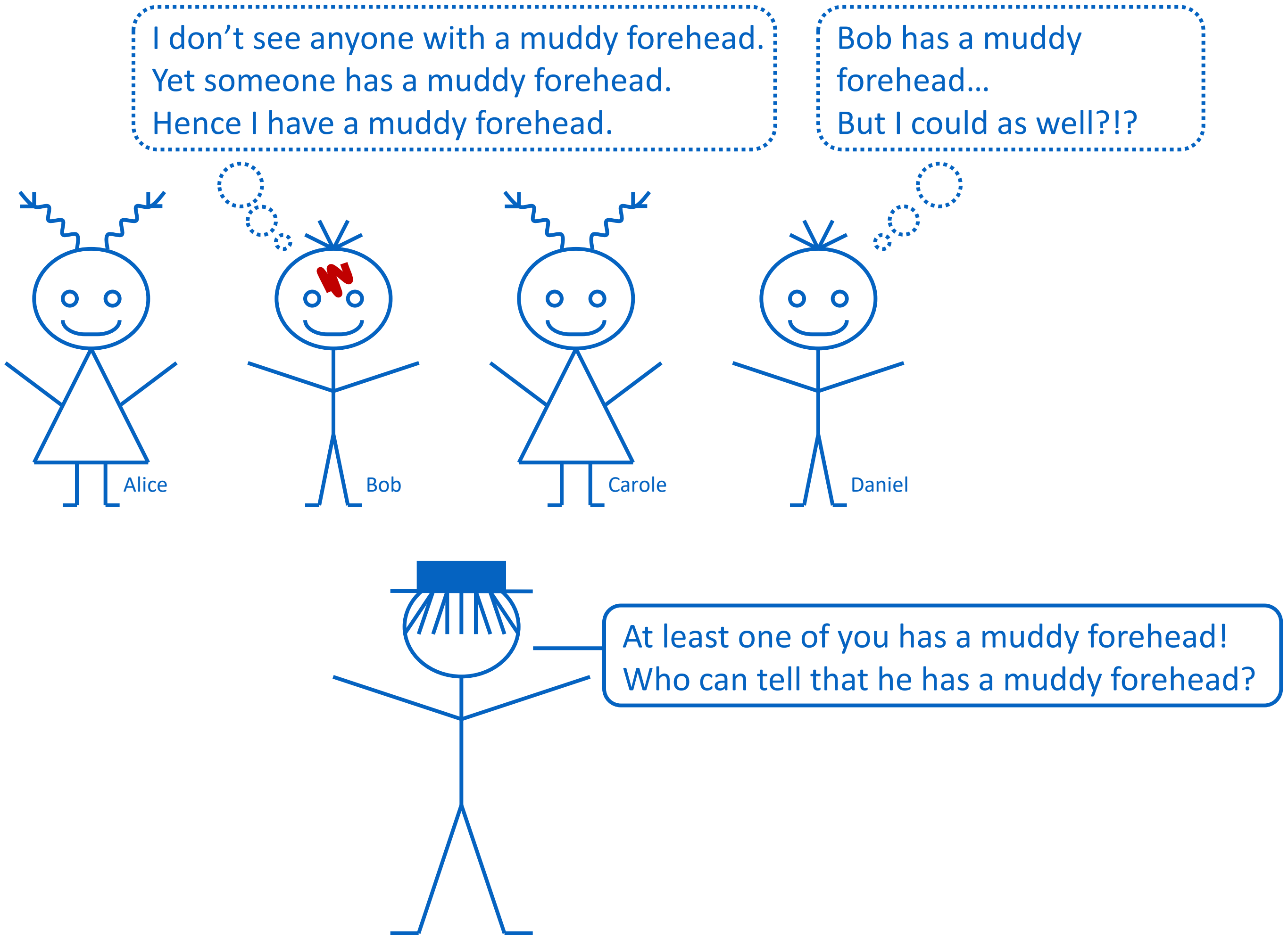
Common knowledge [Halpern et al.]

The "muddy children" puzzle

- *N* children play together,
  k get mud on their foreheads
- Each can see the mud on others but not on his forehead
- The father comes and says:

  "At least one of you has mud on your forehead"
  (a fact known by children)
- The father asks over and over:

  "Does any of you know whether you have mud on your own forehead?"
- All the children are perceptive, intelligent, truthful, and they answer simultaneously



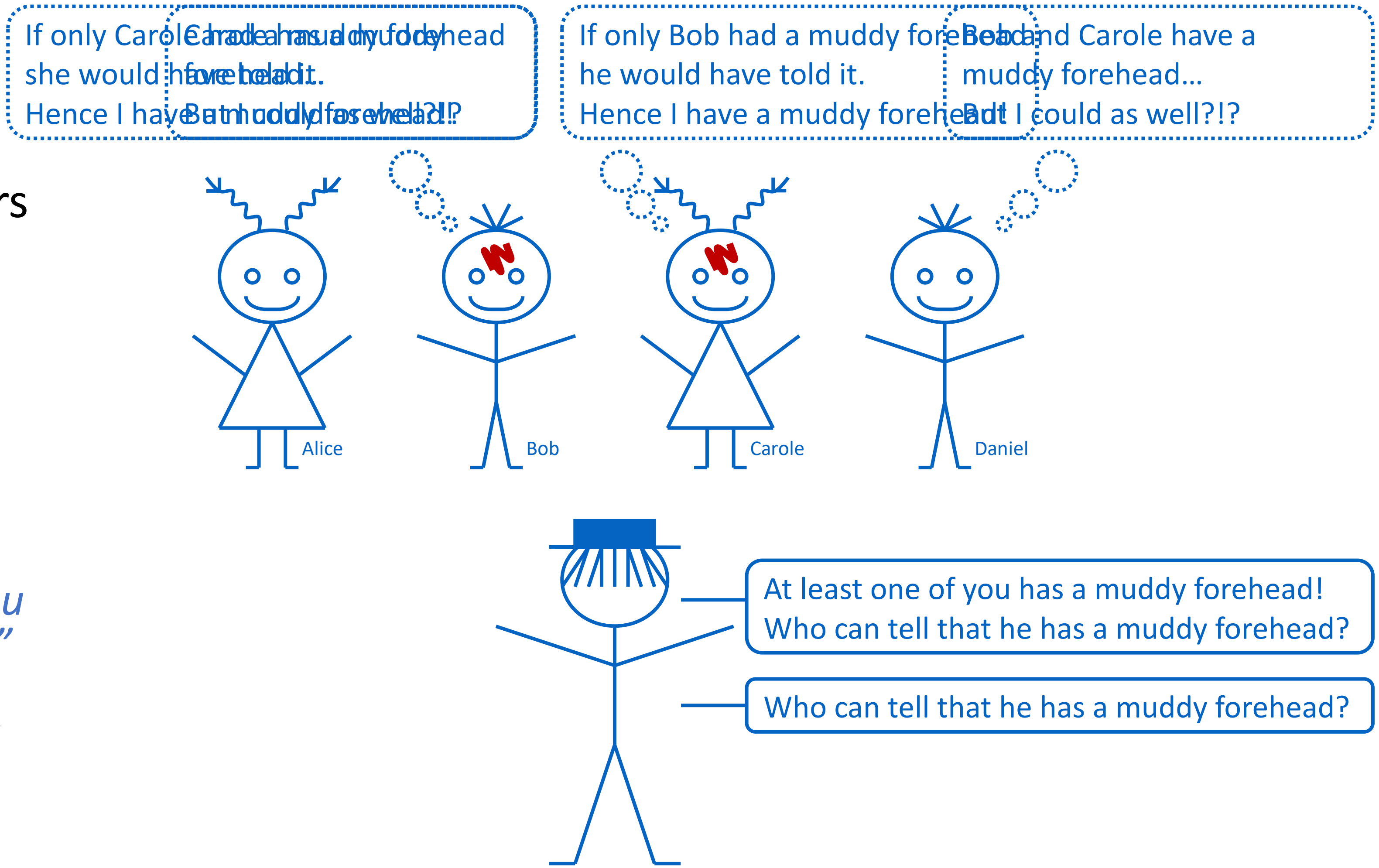
Common knowledge [Halpern et al.]

The “muddy children” puzzle

- *N* children play together, *k* get mud on their foreheads
- Each can see the mud on others but not on his forehead
- The father comes and says:

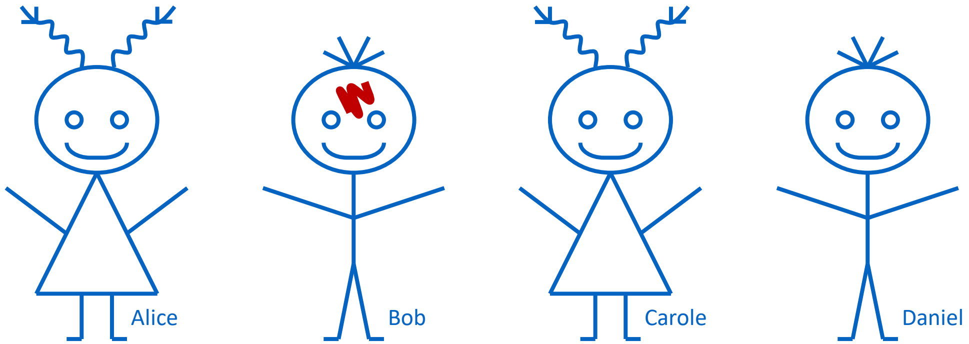
  “At least one of you has mud on your forehead”
  (a fact known by children)
- The father asks over and over:

  “Does any of you know whether you have mud on your own forehead?”
- All the children are perceptive, intelligent, truthful, and they answer simultaneously



Common knowledge [Halpern et al.]

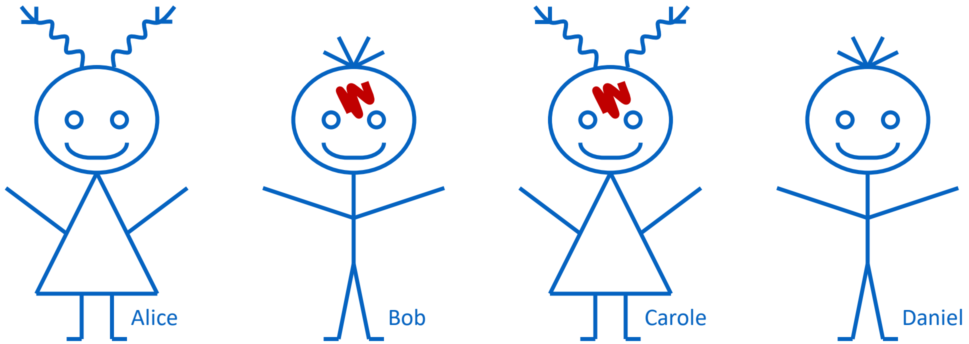
- What's the use of the father's statement
  "At least one of you has a muddy forehead!"



  - Alice, Carole and Daniel know that "at least one children has a muddy forehead"
  - Bob does **not** know that "at least one children has a muddy forehead"

Common knowledge [Halpern et al.]

- What's the use of the father's statement
  "At least one of you has a muddy forehead!"



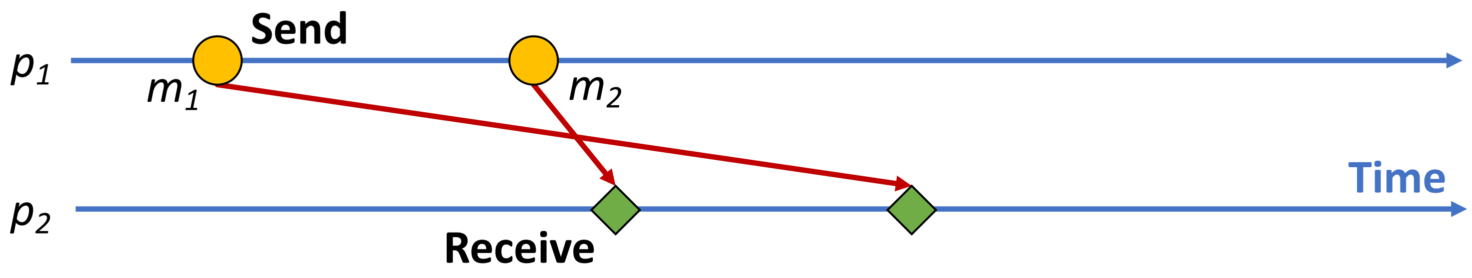
- All know that "at least one children has a muddy forehead"
- Alice and Daniel know that all know that "at least one children has a muddy forehead…
  …but Bob and Carole do **not** know that all know that "at least one children has a muddy forehead"

Algorithms

- An algorithm is a solution to a problem
- Example: **reliable FIFO channels**
- Specification: **reliable channels** + **FIFO**

FIFO: if a process p sends m to q before sending m’ to q, then q does not deliver m’ before m

- Assumption: **reliable channels**



Algorithms

Proof: follows from reliable channel assumption and code

Application

Sender

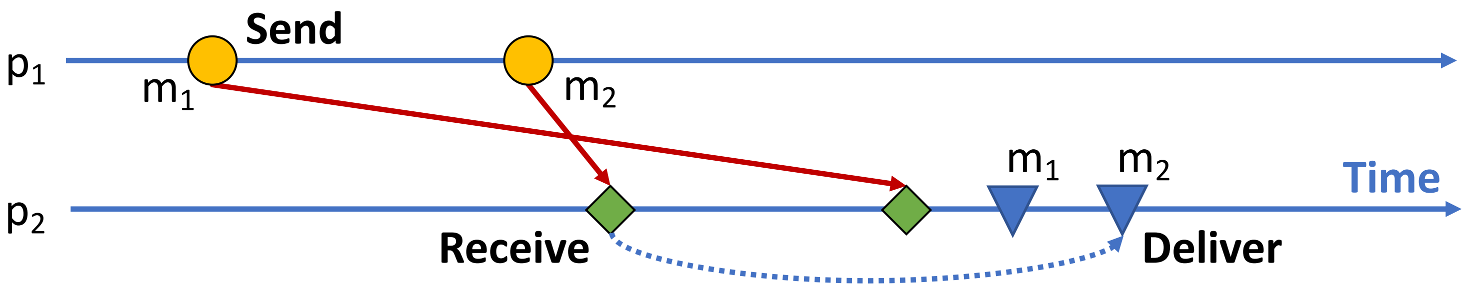
```
s := 0      // Counter (# msg sent)
to send msg
  send (msg,s)
  s := s + 1
```

Reliable channel

Application

Receiver

```
r := 0             // Counter (# msg delivered)
Q := {}   // Queue of (msg,cnt) sorted by cnt
upon receive (msg,cnt)
  enqueue (msg,cnt) in Q
  while (head(Q).cnt = r)
    deliver head(Q).msg
    r := r + 1
    dequeue head(Q)
```



Summary

- To describe a distributed system, must specify system model
  - Communication, failures, synchrony
- Distributed computing problems can be specified by the means of safety and liveness properties…

  ...and solved by distributed algorithms
- It is crucial to be clear and precise about these matters, as they affect whether
  - An algorithm works *in a given system*
  - A computational problem is solvable *in a given system*