Principles of Distributed Computing

Mohsen Lesani

Algorithms

- One processor
  - Reliable (no faults)
  - No communication
- No Concurrency
  - One step at a time
- Complexity
  - Step complexity
- Examples
  - Sorting (Quicksort, Mergesort, Heapsort)
  - Searching (Binary search)
  - Matrix mult. (Strassen's)
  - Primality testing

Distributed Systems

- Many processors
  - Faulty (crash, byzantine, etc.)
  - Communication over network
- Concurrent
  - Multiple steps at a time
- Complexity
  - Message complexity
  - Latency analysis
- Examples
  - Leader election
  - Consensus (Agreement)
  - Mutual exclusion (Dining Philosophers)
  - Atomic objects

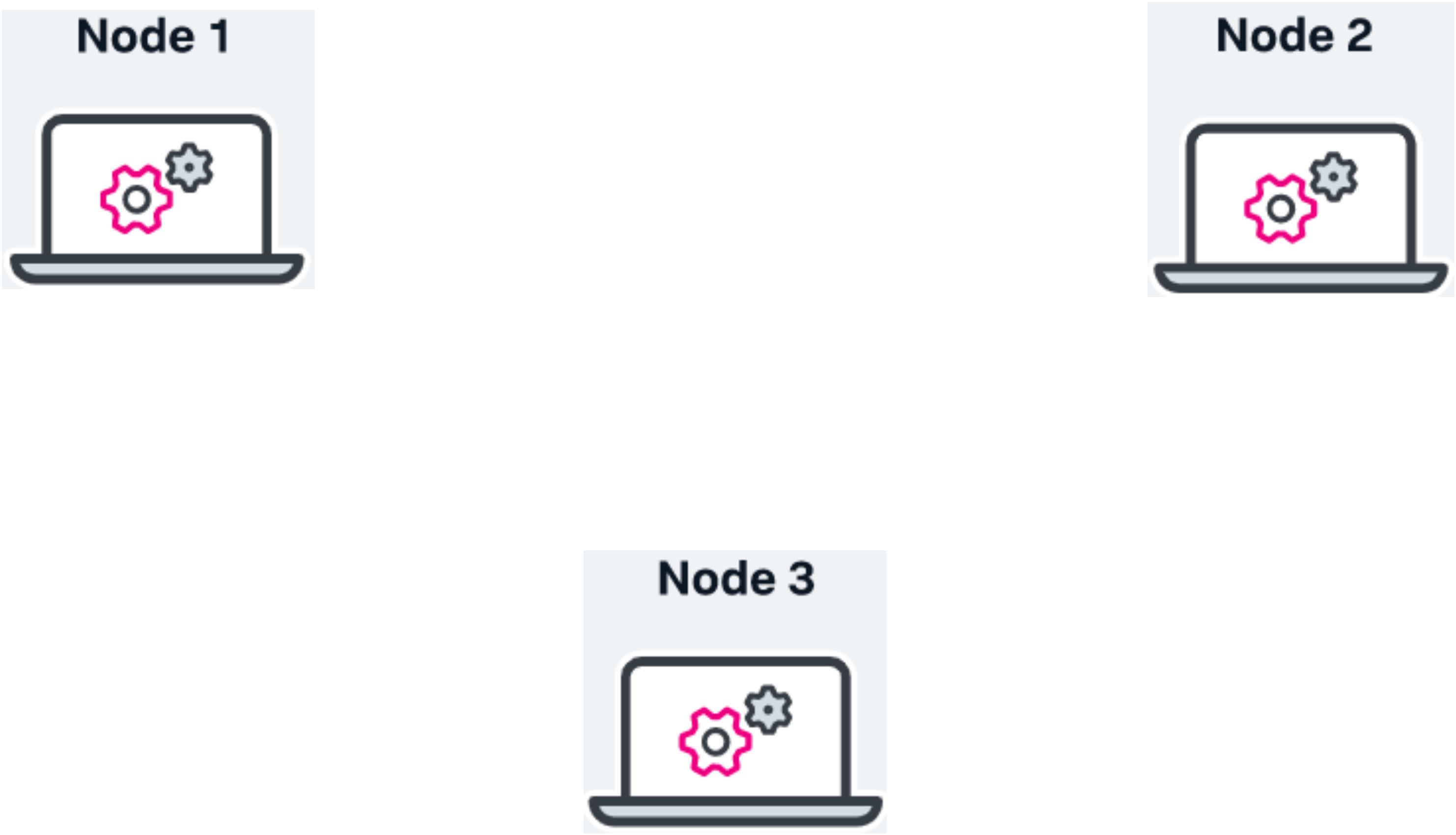
History Lesson

- 1960’s: Edsger W. Dijkstra
  - Concurrent operating systems
  - Semaphores
  - Dining Philosophers (mutual exclusion)
  - Self-stabilization (fault-recovery)
- 1970’s: Leslie Lamport
  - Logical clocks (time and causality)
  - Replication
  - Byzantine Generals Problem (consensus)
  - “A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.”
- 1970's: Jim Gray
  - Transactions
  - Databases
- 1980's: Nancy Lynch
  - Fault tolerance
  - Timing (synchrony, asynchrony, partial synchrony)
  - Consensus
- 1990's: Birman, Schneider, Toueg
  - Failure detectors
  - Reliable broadcast
  - Totally-ordered broadcast
  - Causal broadcast
  - Group membership
  - View synchrony
- 2020’s
  - Map Reduce, Google File System
  - Raft, Spanner
  - Spark
  - Bitcoin

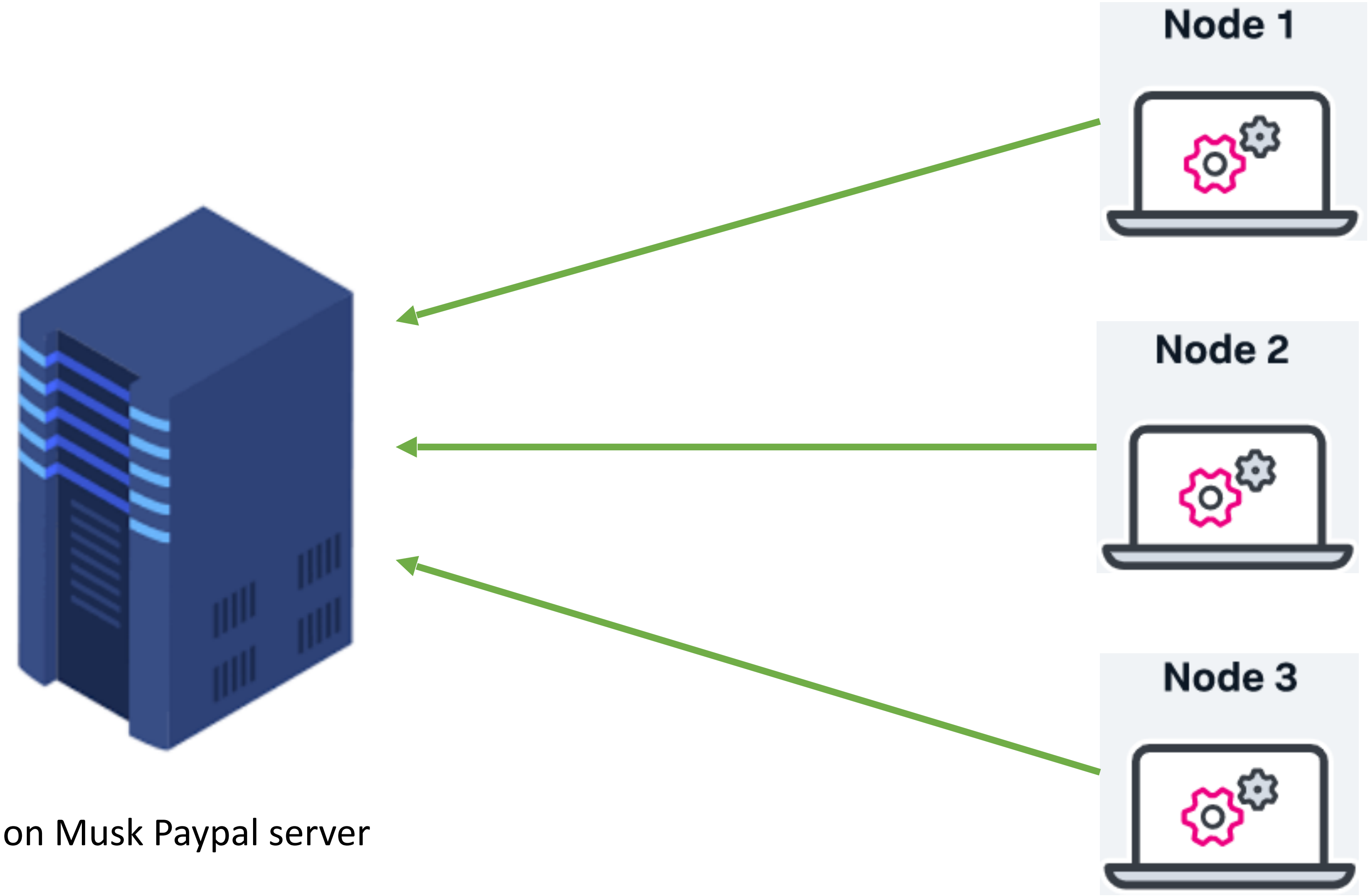
Todays Lecture

- Big picture:
  - What is a distributed system?
  - Why build a distributed system?
- Components of a distributed system:
  - Processes (abstracting computers)
  - Channels (abstracting networks)
- Time & failure detectors

A distributed system

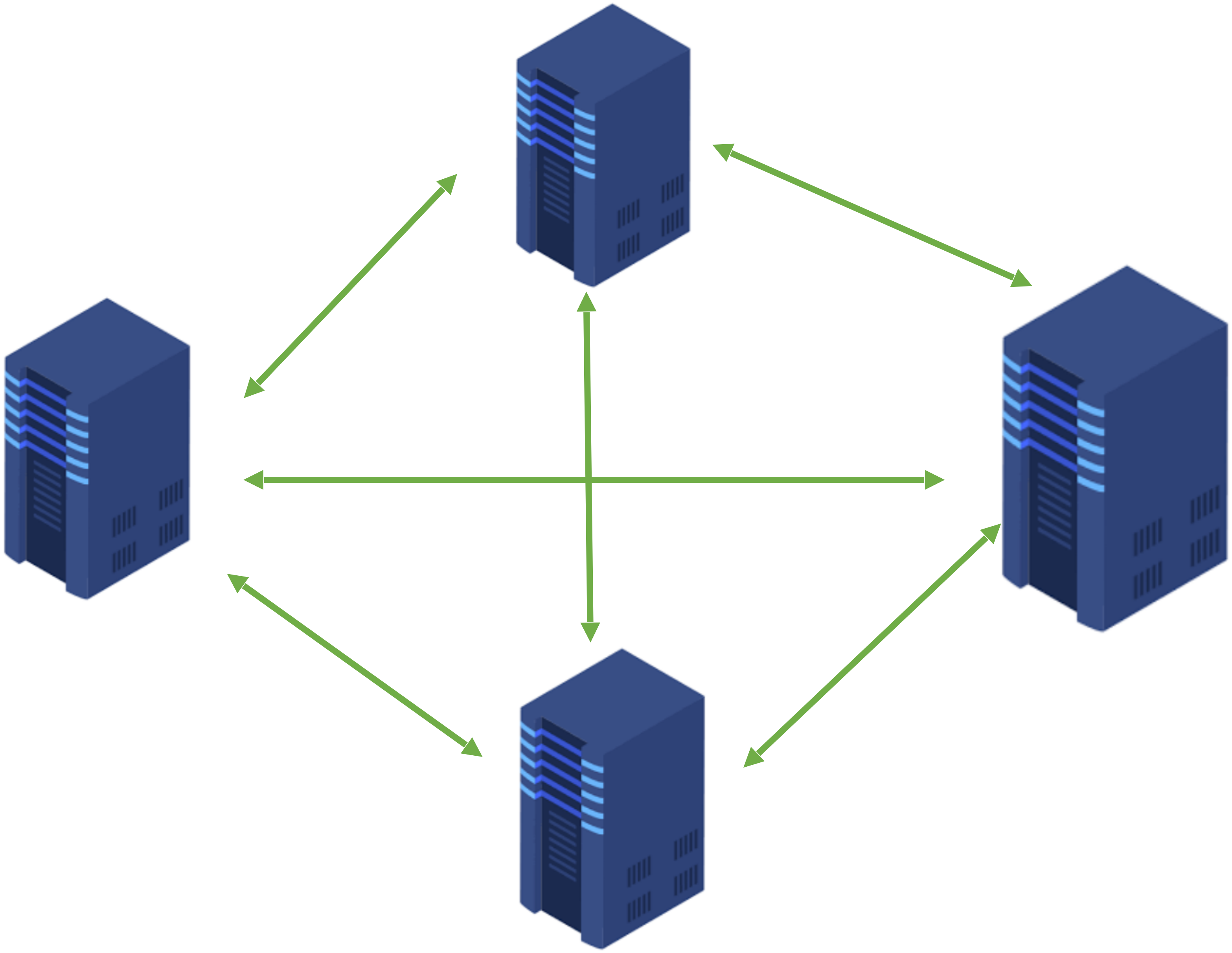


Client-server system



Elon Musk Paypal server

Multiple Servers



Why distributed systems?

- What are the advantages?

Distributed	vs.	Centralized
Multi-server		Client-server

Why distributed systems?

- What are the advantages?

Distributed	vs.	Centralized
Multi-server		Client-server

- High-availability / Fault-tolerance
- Locality, Responsiveness
- Concurrency / Parallelism -> Performance

Why not distributed systems?

- What are the disadvantages?

Distributed	vs.	Centralized
Multi-server		Client-server

Why not distributed systems?

- What are the disadvantages?

Distributed	vs.	Centralized
Multi-server		Client-server

- Expensive (to have redundancy)
- Concurrency -> Interleaving -> Bugs
- Failures -> Incorrectness

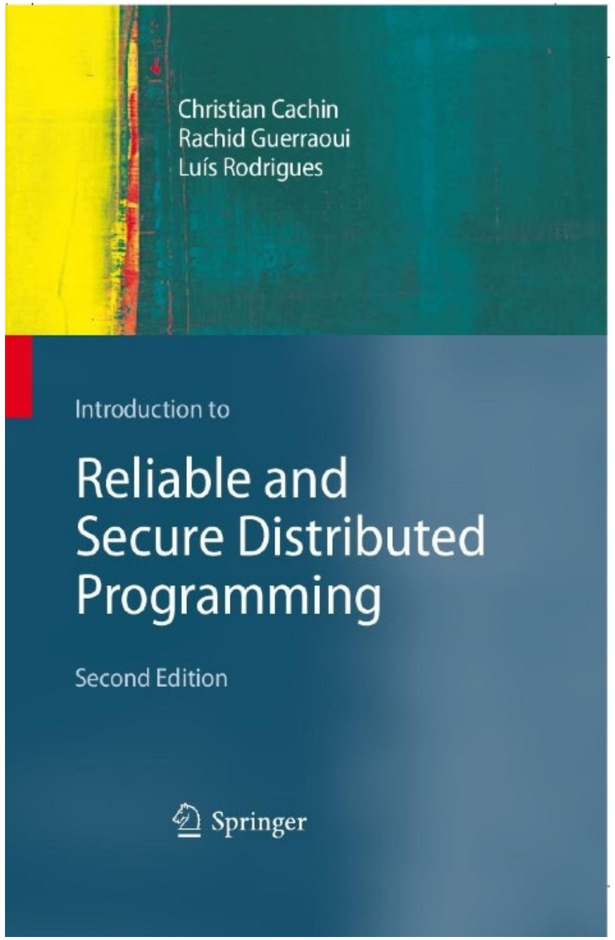
Todays Lecture

- Big picture:
  - ✓What is a distributed system?
  - ✓Why build a distributed system?

- Components of a distributed system:
  - Processes (abstracting computers)
  - Channels (abstracting networks)
  - Time & failure detectors

- Time & failure detectors

Textbook



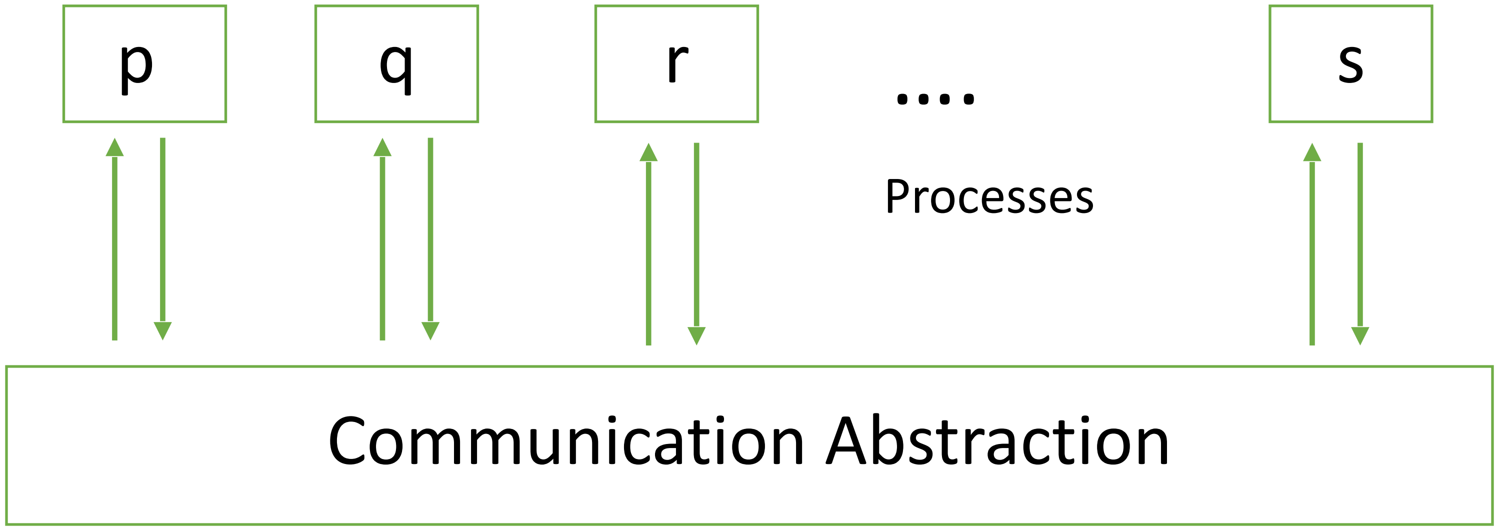
Introduction to Reliable and Secure Distributed Programming

C. Cachin, R. Guerraoui, L. Rodrigues

2nd ed. of "Introduction to Reliable Distributed Programming"

The new content covers Byzantine failures.

Distributed Programming

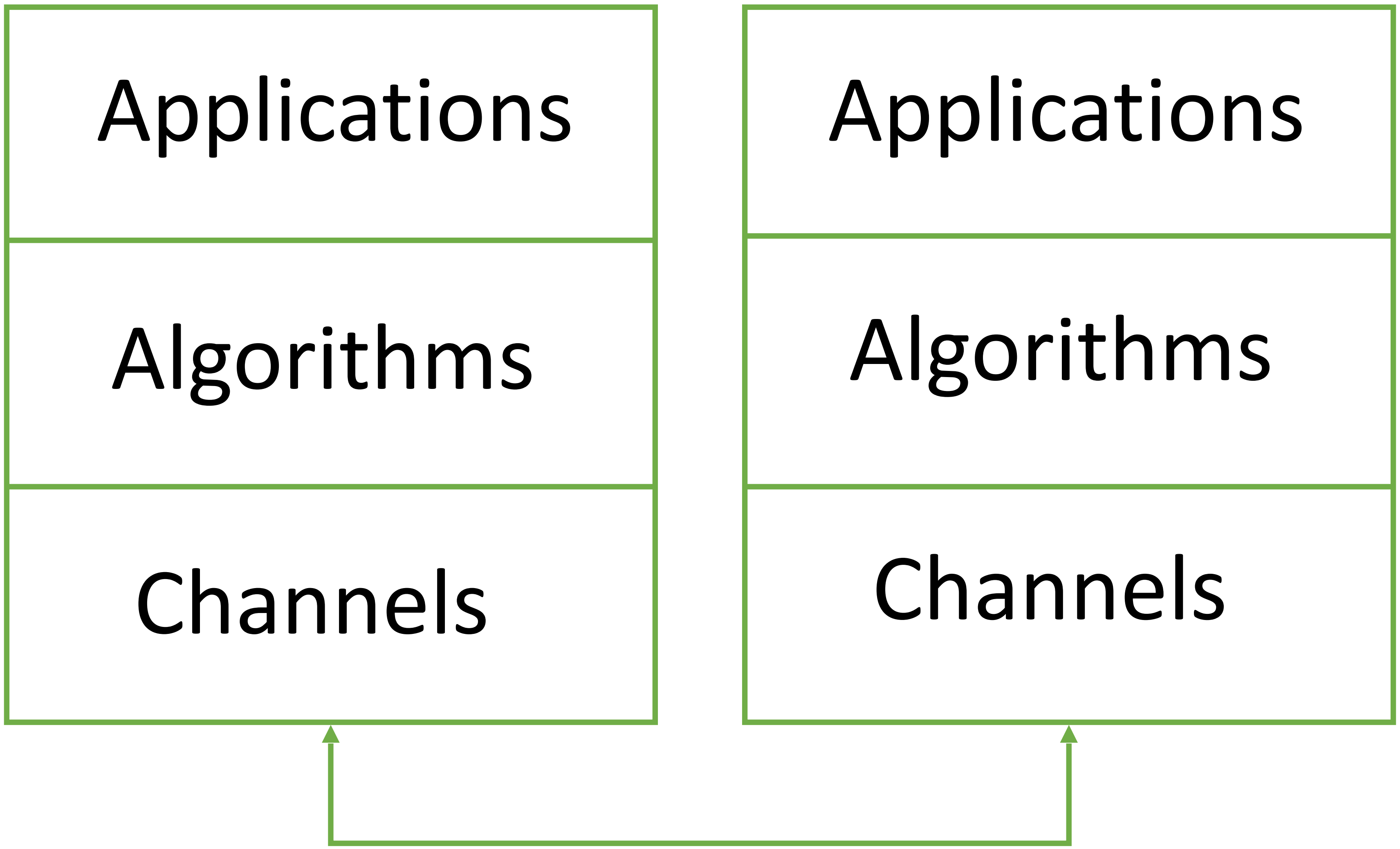


- System with N processes (also called replicas) $\Pi = \{p, q, r \ldots\}$. (Processes know each other.)
- Processes coordinate to implement the application

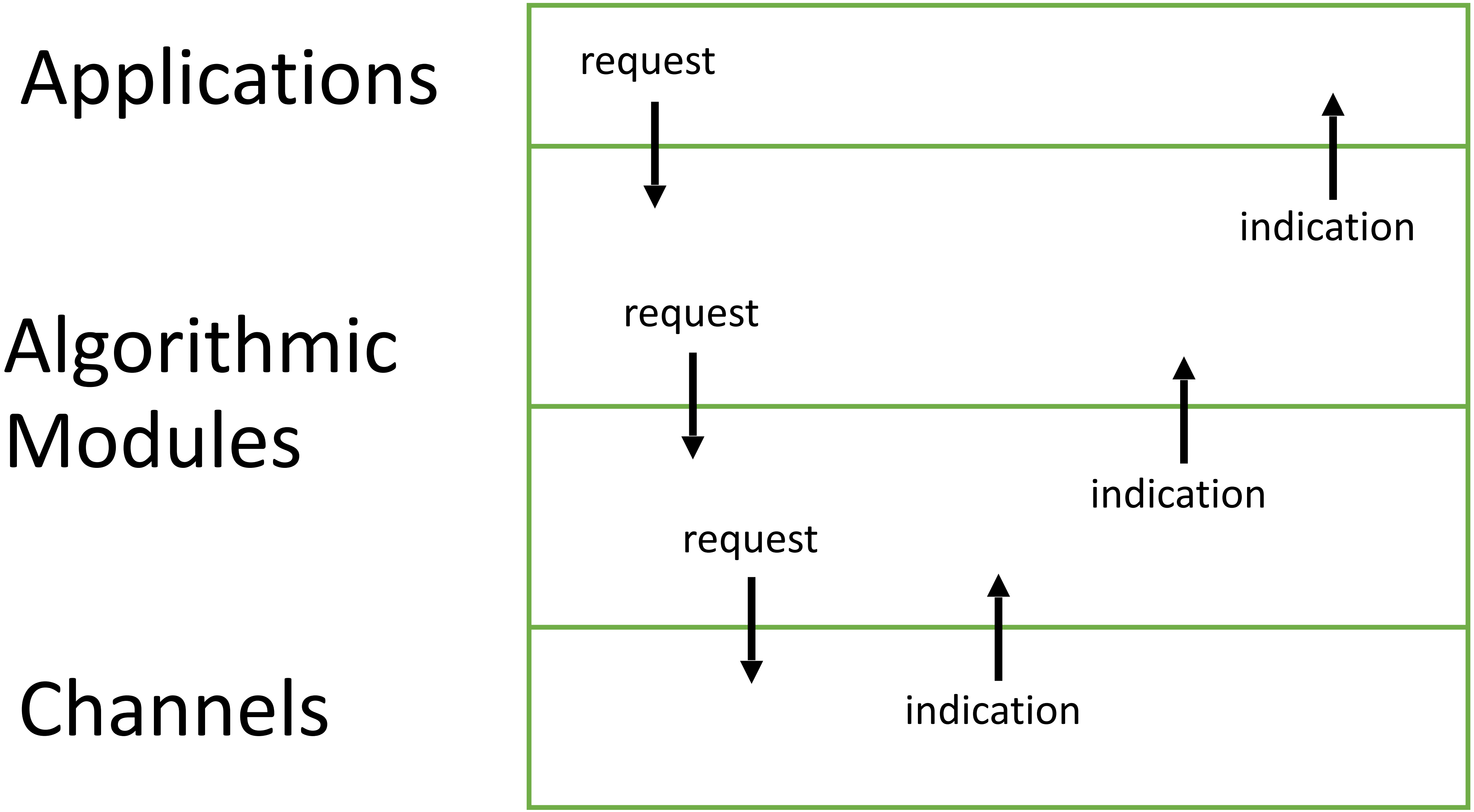
Programming abstractions

- Sequential programming
  - Array, record, list ...
- Concurrent programming
  - Thread, semaphore, monitor, ...
- Distributed programming
  - Reliable broadcast
  - Shared memory
  - Consensus
  - Atomic commit
  - ...

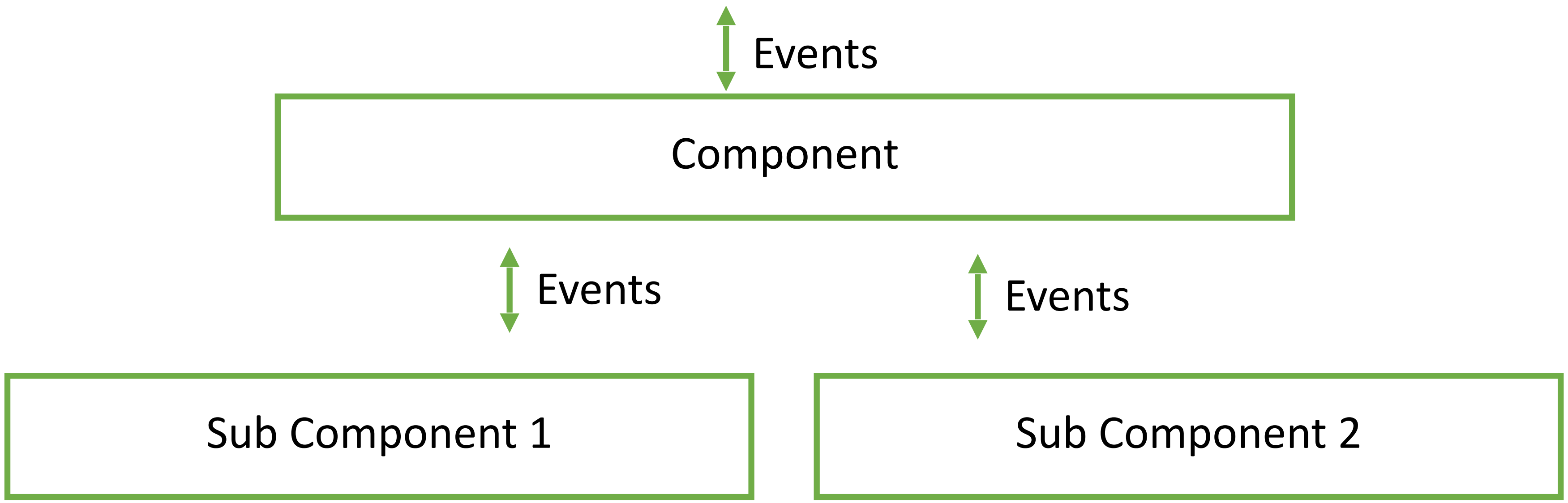
Distributed System



Modules of a process

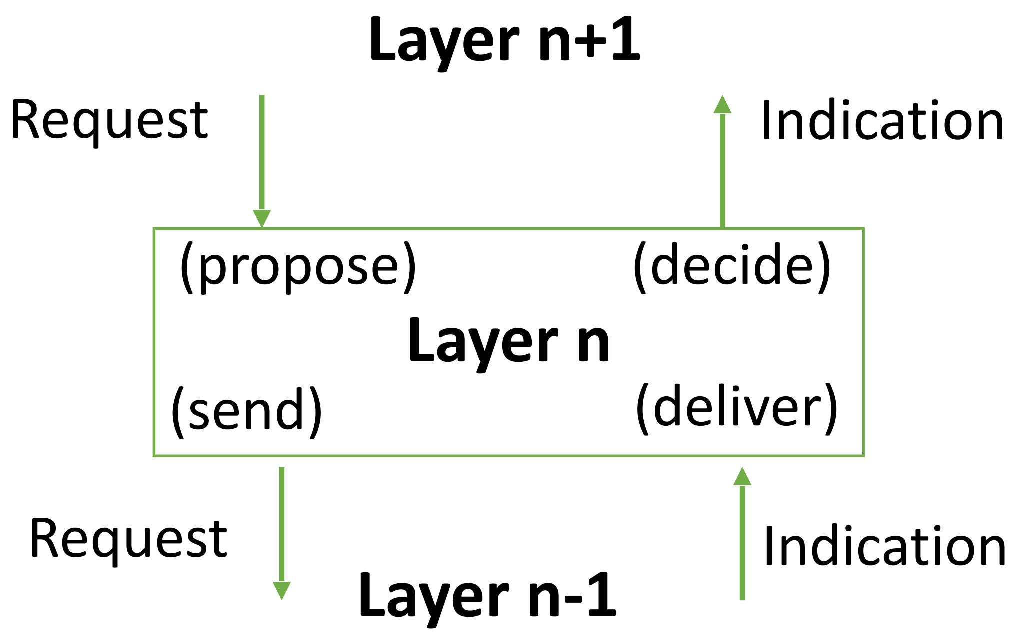


Layered Modular Architecture



- Every process is a tree of components.
  - Every component has a unique identifier.
  - There might be multiple instances of a component type.
- Modules communicate through events.

Programming with Events



- Asynchronous events
  - Request events flow downward
  - Indication (or Response) events flow upward

Reactive Programming

A component is implemented as a set of event handlers

upon event <component, Event (att1, att2 …) > **do**
    do something;
    trigger <component', Event' (att’1, att'2 …) >;

The component is elided if it is the current component **self**.

Specification

What does a component provide?

Specification in terms of the interface events.

Example Components

- Reliable broadcast
  - Ensure that a message sent to a group of processes is received by all or none.
- Atomic commit
  - Ensure that the processes reach a common decision on whether to commit or abort a transaction.

Module Specification

- A module is defined by events and properties:

Reliable Broadcast

- Events
  - Request: < broadcast (m) >
  - Indication: < deliver (src, m) >
- Properties:
  - Validity
  - No Duplication
  - No creation
  - Agreement

Module Specification

- A module is defined by events and properties:

Atomic Commit

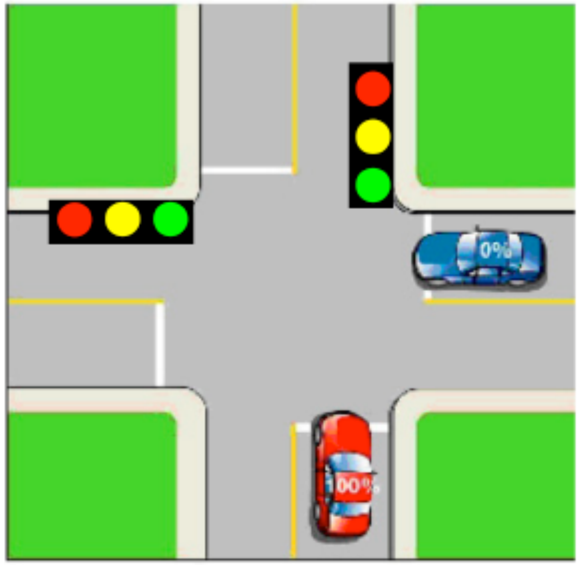
- Events
  - Request: < propose (d) > where d is either Commit or Abort
  - Indication: < decide (d) >
- Properties:
  - Uniform Agreement
  - Integrity
  - Abort Validity
  - Commit Validity
  - Termination

Two Types of Properties

- **Safety** properties state that nothing bad ever happens.
- **Liveness** properties state that something good eventually happens.

Safety and Liveness

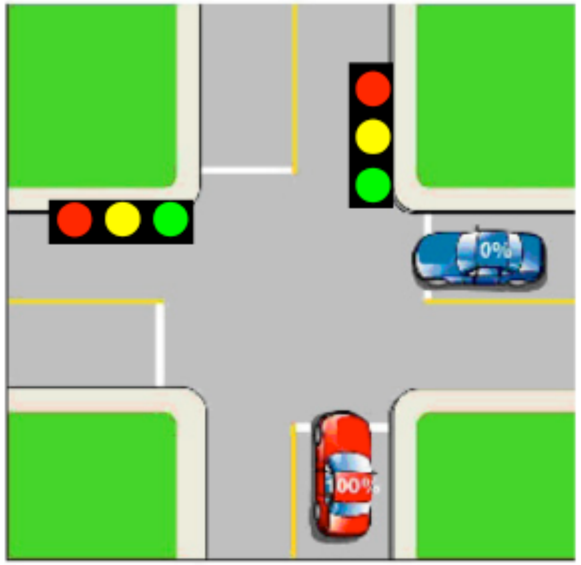
- Example: Traffic lights



- Only one direction gets a green light

Safety and Liveness

- Example: Traffic lights



- Eventually each direction gets a green light

Safety and Liveness

- Example: Reliable Broadcast
  Eventually every message is delivered.

Safety and Liveness

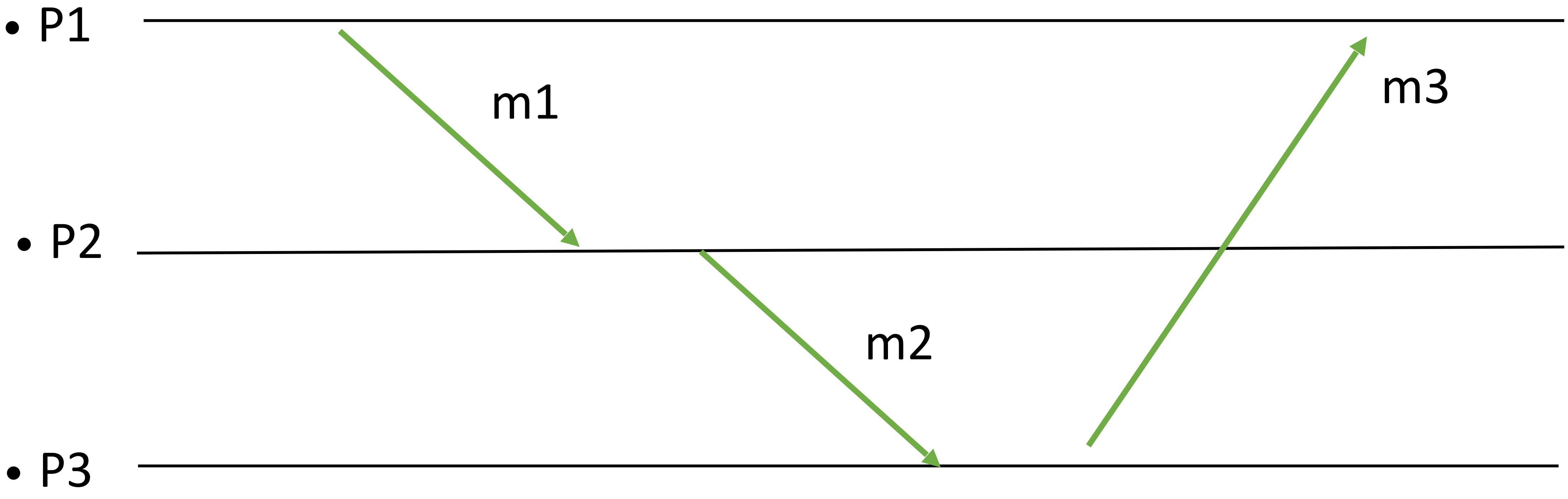
- Example: Failure Detector

  Strong Completeness: Eventually, every process that crashes is permanently suspected by every correct process.

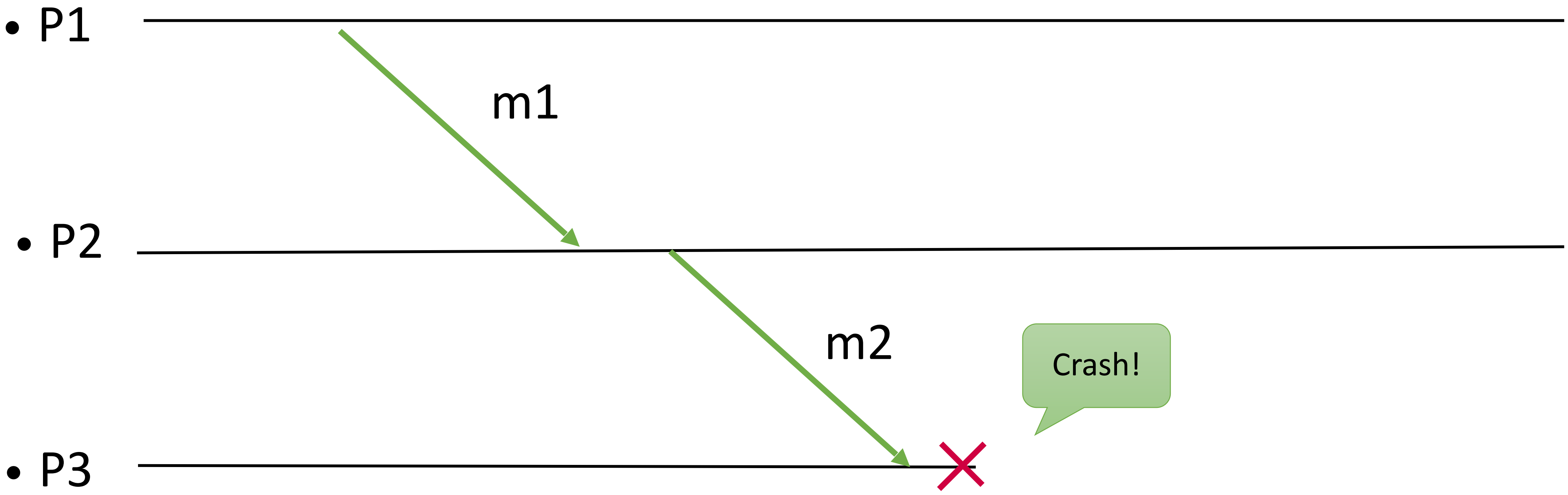
Safety and Liveness

- Example: Failure Detector
  Strong Accuracy: No process is suspected before it crashes.

Execution Traces



Execution Traces



Processes

Processes may fail:

- **Crash-stop**: The process takes no further process.

  Simply a more specific case of emissions (dropping messages):If a process omits a message, then it omits all subsequent messages.

- **Arbitrary (Byzantine)**: The process can take arbitrary including malicious actions. For example, it can send misleading messages.

A process that does not fail is called **correct.**

A process that fails is called **incorrect**.

Processes

- By default, we assume crash-stop processes.
  - Processes fail only by crashing.
  - Processes do not recover.

Todays Lecture

- Big picture:
  - ✓ What is a distributed system?
  - ✓ Why build a distributed system?

- Components of a distributed system:
  - ✓ Processes (abstracting computers)
  - Channels (abstracting networks)
  - Time & failure detectors

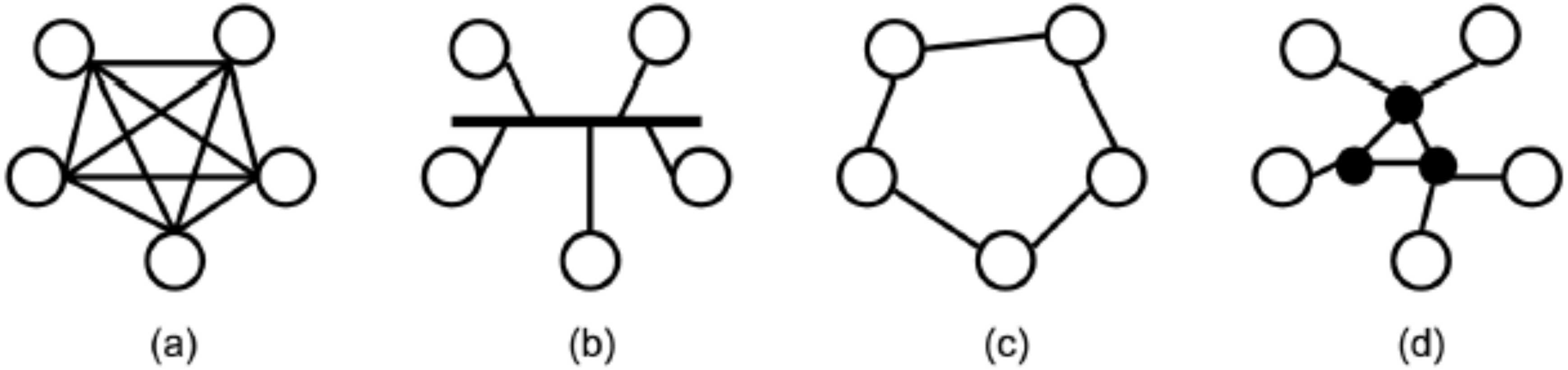
- Time & failure detectors

Channels

- Processes communicate by message passing through communication channels.
- We consider point-to-point channels.
- Messages are uniquely identified and the message identifier includes the sender's identifier.

Links

- Logically every process may communicate with every other process: (a)
- Physical implementation may differ: (b)-(d)



Channels

How reliable are the communication channels?

- Fair-loss links:
  - Messages may be lost, but is delivered with some small probability.
- Stubborn links:
  - Eventually messages delivered (infinitely often).
- Perfect links:
  - Eventually each message is delivered once.

Channel Specification

A channel module is defined by events and properties:

- Events
  - Request: send (dest, m)
  - Indication: deliver (src, m)
- Properties:
  - Reliability
  - No Duplication
  - Integrity
  - …

Fair-loss links

- FL1. Fair-loss:
  - If a message is sent infinitely often by pi to pj, and neither pi or pj crash, then m is delivered infinitely often to pj.
- FL2. Finite duplication:
  - If a message is sent a finite number of times by pi to pj, it is not delivered an infinite number of times to pj.
- FL3. No creation:
  - No message is delivered unless it was sent.

Stubborn links

- SL1. Stubborn delivery.
  - If a correct process pi sends a message m to a correct process pj, then pj delivers m, an infinite number of times.
- SL2. No creation:
  - No message is delivered unless it was sent.

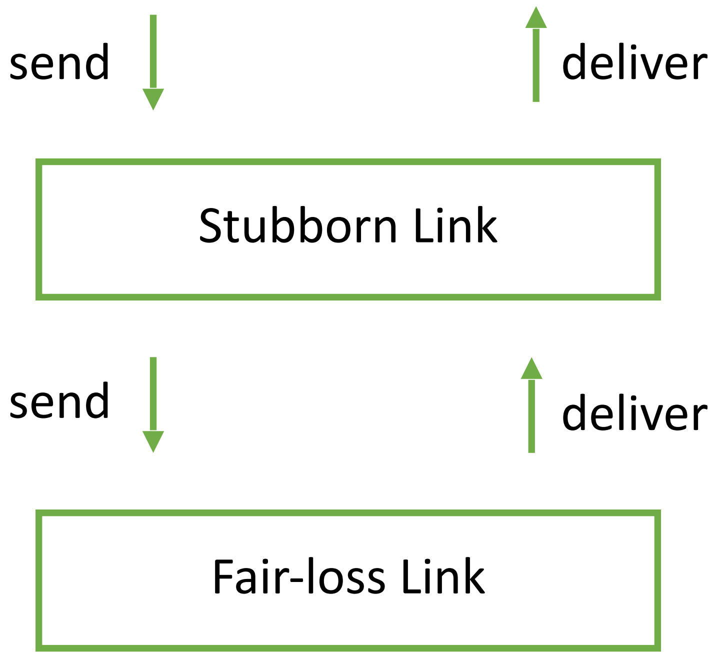
Algorithm (sl)

Implements: StubbornLinks (sl)

Uses: FairLossLinks (fl)

upon event <sl, send (dest, m)> **do**
    repeat forever
        trigger <fl, send (dest, m)>

upon event <fl, deliver (src, m)> **do**
    trigger <sl, deliver (src, m)>



Reliable (Perfect) links

- PL1. Validity.
  - If pi and pj are correct, then every message sent by pi to pj is eventually delivered by pj.
- PL2. No duplication:
  - No message is delivered to a process more than once.
- PL3. No creation:
  - No message is delivered unless it was sent.

Algorithm (pl)

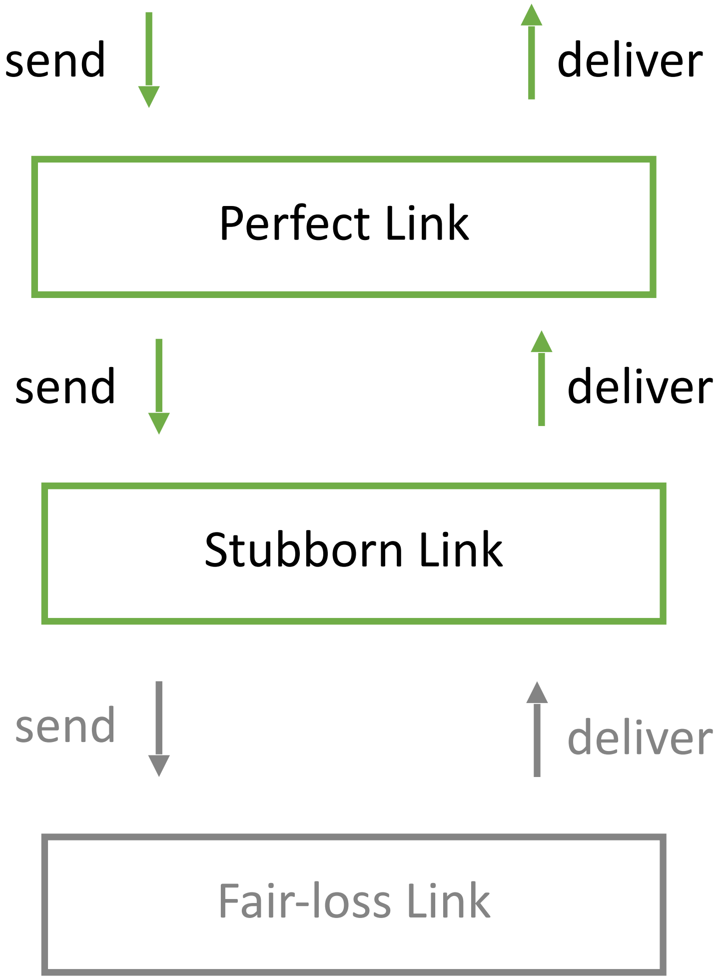
Implements: PerfectLinks (pl)

Uses: StubbornLinks (sl)

upon event < Init > **do** delivered := ∅

upon event < pl, send (dest, m) > **do**
  trigger < sl, send (dest, m) >

upon event < sl, deliver (src, m) > **do**
  if m ∉ delivered **then**
    trigger < pl, deliver (src, m) >
    delivered := delivered ∪ {m}



Reliable links

- We implicitly assume perfect links.
- Roughly speaking, reliable links ensure that messages exchanged between correct processes are not lost.

Todays Lecture

- Big picture:
  - ✓ What is a distributed system?
  - ✓ Why build a distributed system?

- Components of a distributed system:
  - ✓ Processes (abstracting computers)
  - ✓Channels (abstracting networks)
  - Time & failure detectors

- Time & failure detectors

Time

- Local clocks:
  - Do processes have access to local clocks?
  - If so, are these clocks synchronized? Are these clocks accurate?
    - clock skew — Difference between time
    - clock drift — Difference between clock rate
- Communication channels:
  - How long does a message take to be delivered?

Models of Synchrony

Synchrony: perfectly synchronized rounds

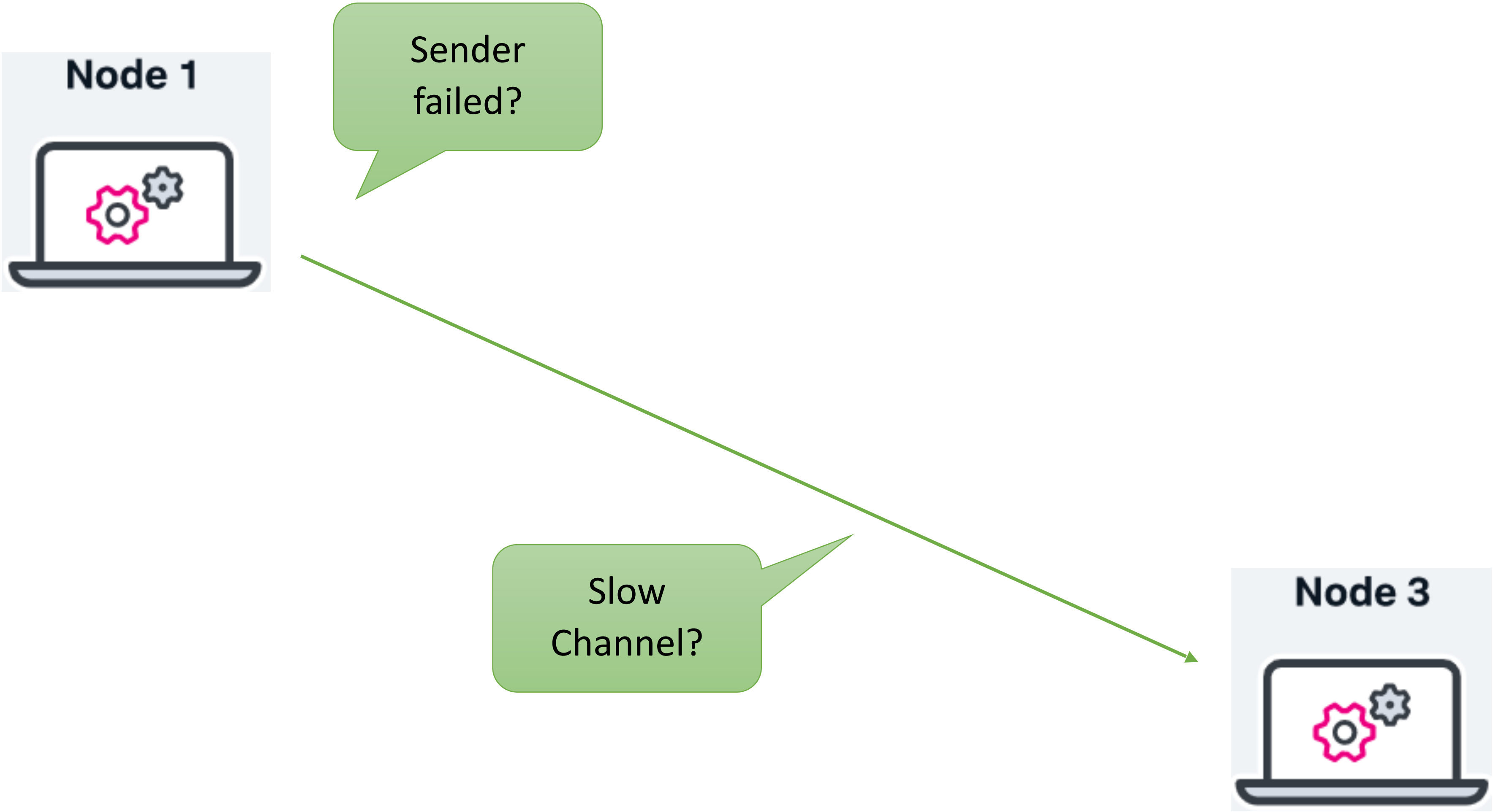
Partial Synchrony

Asynchrony: anything goes

Timing assumptions

- Synchronous:
  - Processing: the time it takes for a process to execute a step is bounded and known.
  - Delays: there is a known upper bound limit on the time it takes for a message to be received.
  - Clocks: the drift between a local clock and the global real time clock is bounded and known.
- Eventually Synchronous:
  - Synchronous timing holds eventually.
- Asynchronous:
  - No assumptions, no clocks.

Time and Failure Detection



Failure Detector

- A failure detector is a distributed component that provides processes with suspicions about crashed processes.
- It is implemented using (i.e., it encapsulates) timing assumptions.
- According to the timing assumptions, the suspicions can be accurate or inaccurate.

Failure Detector

Failure detector component

- Events
  - Indication: < crash (p) >
  - Indication: < restore (p) >
- Properties:
  - Completeness
  - Accuracy

Failure Detector

- Perfect:
  - Strong Completeness: Eventually, every process that crashes is permanently suspected by every correct process.
  - Strong Accuracy: No process is suspected before it crashes.
- Eventually Perfect:
  - Strong Completeness
  - Eventual Strong Accuracy: **Eventually**, no correct process is ever suspected.

Failure Detector

Implementation:

- Processes periodically exchange heartbeat messages.
- A process sets a timeout based on worst case roundtrip of a message exchange.
- A process suspects another process if its times out is triggered.
- A process that receives a message from a suspected process revises its suspicion and increases its timeout.

Failure Detectors

Network model:

- Synchronous ->
- Eventual Synchronous ->
- Asynchronous ->

Guarantees:

Failure Detectors

Network model:		Guarantees:
• Synchronous	->	Perfect FD
• Eventual Synchronous	->	
• Asynchronous	->	

Failure Detectors

Network model:		Guarantees:
• Synchronous	->	Perfect FD
• Eventual Synchronous	->	Eventually Perfect FD
• Asynchronous	->	

Failure Detectors

Network model:		Guarantees:
• Synchronous	->	Perfect FD
• Eventual Synchronous	->	Eventually Perfect FD
• Asynchronous	->	None!!

Protocol Design

Assumptions:

- Processes: crash-stop failures
- Channels: reliable channels
- Timing: perfect OR eventually perfect failure detectors

For every service:

- We develop algorithms for a crash-stop system with a perfect failure detector.
- We try to make a weaker assumptions and revisit the algorithms.

Cryptographic primitives

Dual goals of cryptography

- Confidentiality (encryption, not relevant here)
- Integrity
  - Hash functions
  - Message authentication codes (MAC)
  - Digital signatures

Hash functions

- Cryptographic hash function H maps inputs of arbitrary length to a short unique hash value.
- Collision-freedom: No process can find distinct values x and x' such that H(x) = H(x')

Message-Authentication Codes

- A MAC authenticates data between two processes
- It is based on a shared symmetric key, which is known only to the sender and to the receiver of a message, but to nobody else.
- For a message of its choice, the sender can compute an authenticator for the receiver. Given an authenticator and a message, the receiver can verify that the message has indeed been authenticated by the sender.
- Symmetric cryptographic can be computed and verified quickly.

Digital signatures

- Digital signatures are based on public-key cryptography (or asymmetric cryptography).
- The sender owns a private key that must remain secret; the public key is accessible to anyone. With the private key, the sender can produce a signature for a message.
- Everyone with access to the public key can verify that the signature on the message is valid.
- A signature scheme is more powerful than a MAC in the sense that if a relayed message is verified, only the owner of the private key can be the sender.
- Because of their underlying mathematical structure, asymmetric cryptography adds considerable computational overhead compared to symmetric cryptography.