Overview

- Motivation:
  - Why study fault-tolerant distributed algorithms?
  - Where are they used?
- Formalizing distributed algorithms
  - Basic abstractions: processes and channels
  - The software stack
- How should we study fault-tolerant distributed algorithms?
  - Specifications: safety and liveness
  - Assumptions about processes, channels, and their failures
  - Example: building perfect communication links
  - **Assumptions about timing**
  - Depicting algorithm runs
- Distributed algorithms HOWTO (summary)

Timing Assumptions

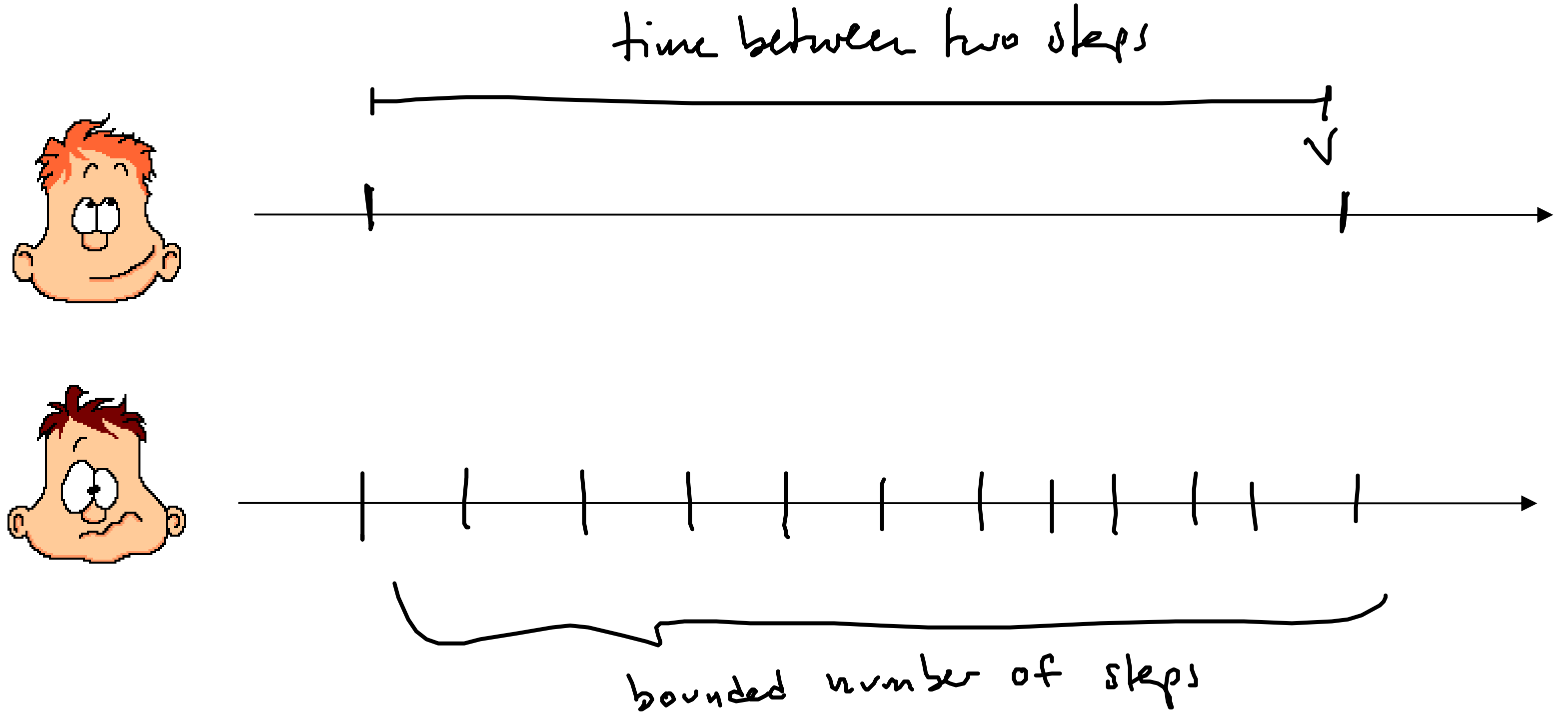
- Timing assumptions relate to
  - different processing speeds (process asynchrony) of processes
  - different speeds of messages (channel asynchrony)
- Three basic types of systems:
  - Asynchronous system
  - Synchronous system
  - Partially synchronous system

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
- ***Asynchronous:*** no assumption
- ***Eventually Synchronous:*** the timing assumptions hold eventually

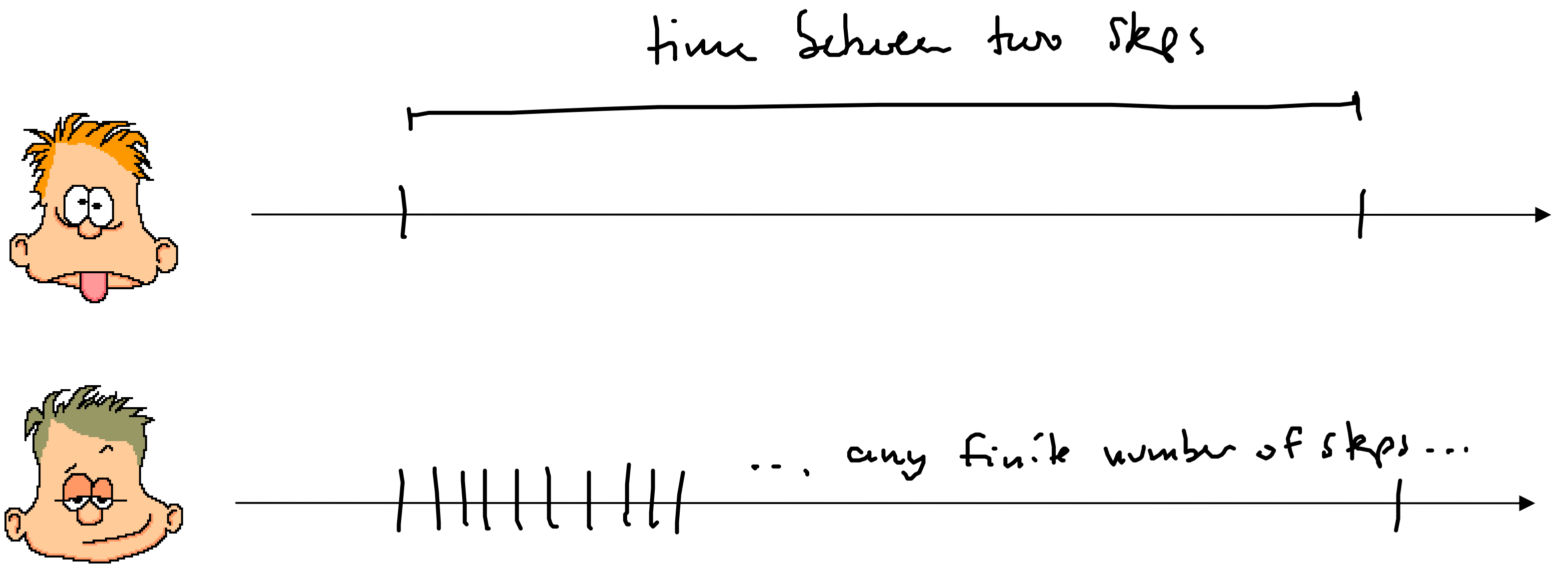
Synchronous System

- While one process takes one step, another process can take at most a bounded number of steps



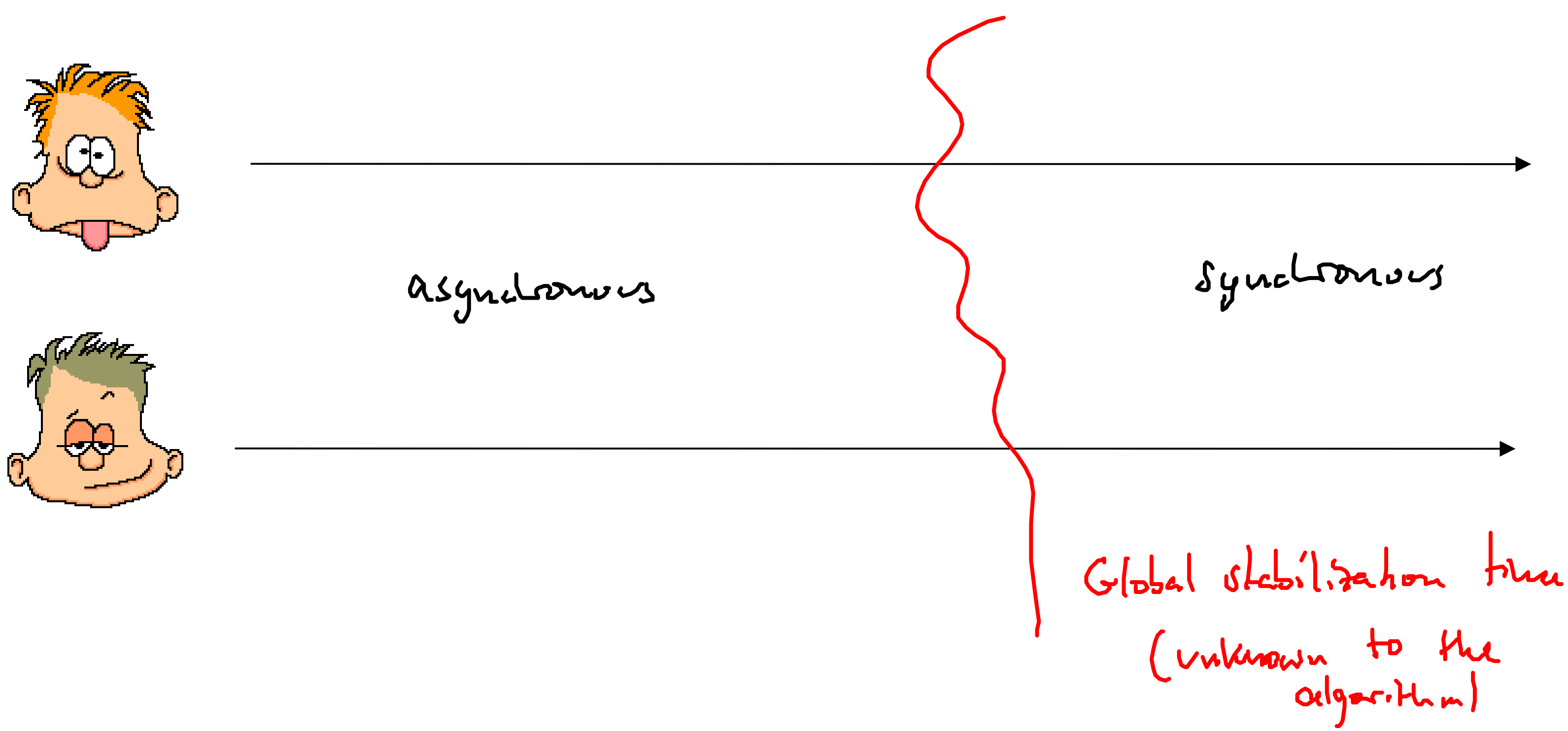
Asynchrony

- While one process takes one step, another process can take any unbounded (but finite) number of steps



Partial Synchrony

- Eventually the system will be synchronous
- Timing bounds hold eventually (but you never know when)



Timing Assumptions

- Timing assumptions are often cumbersome to handle
- Better abstraction: **failure detector**
- Failure detector encapsulated timing assumptions
- Why failure detector?
  - Timeouts are usually used for detecting failures

Failure detection

- A ***failure detector*** is a distributed oracle 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 not

Perfect failure detector

- Indication event: <crash, p>
  Used to notify that process p has crashed
- Properties:
  - ***PFD1***: Eventually every process that crashes is permanently detected by every correct process (strong completeness).
  - ***PFD2*** : No process is detected by any process before it crashes (strong accuracy).

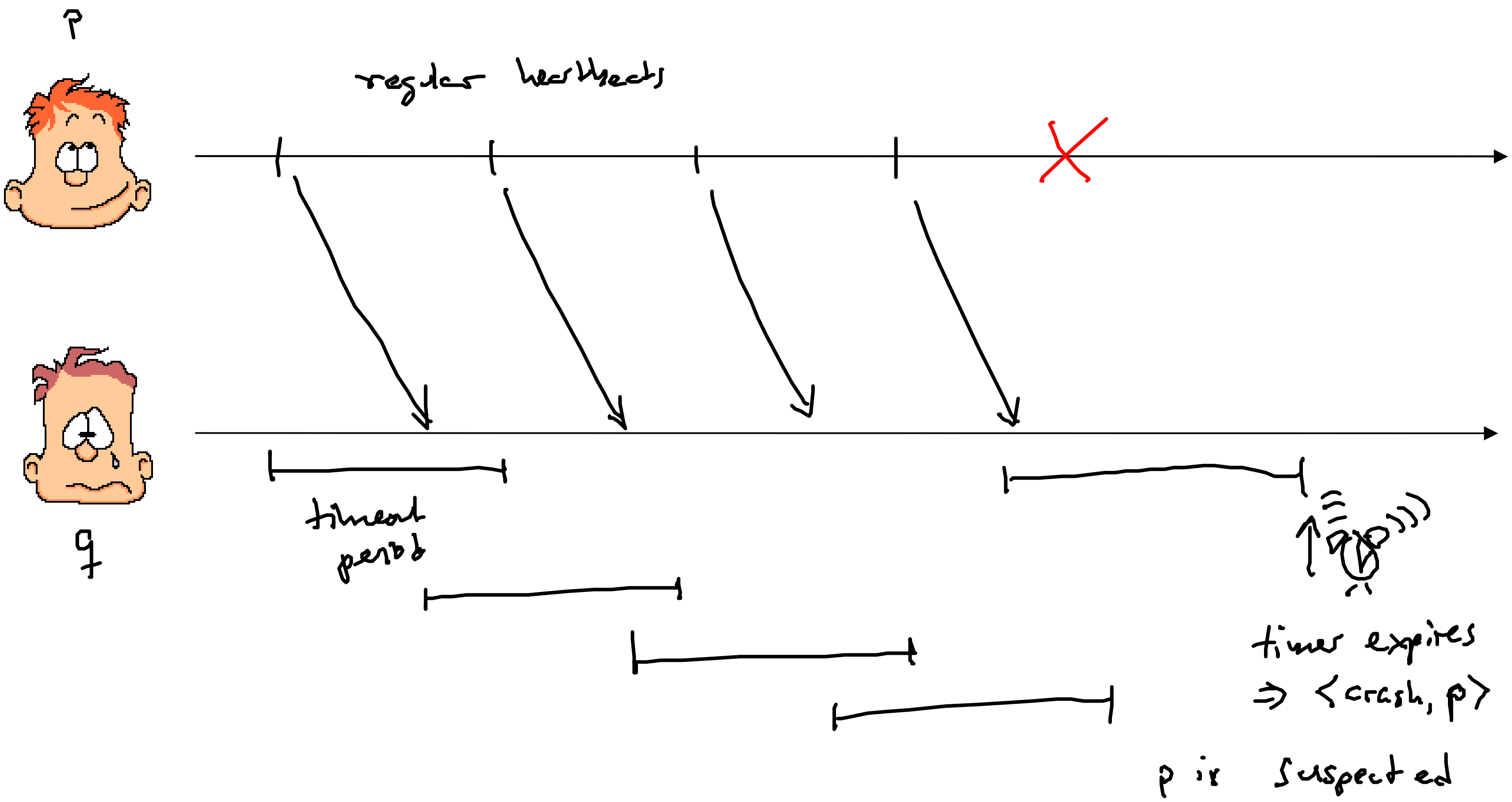
Failure detection

- Implementation:
  - (1) Processes periodically exchange heartbeat messages
  - (2) A process sets a timeout based on worst case round trip of a message exchange
  - (3) A process suspects another process if it timeouts that process
  - (4) A process that delivers a message from a suspected process revises its suspicion and increases its time-out

Formal Algorithm

- **upon** <init> **do**
  - timeout[1..n] = d
  - initialize timer for every process q using timeout[q]
  - suspected = { }
- **periodically do**
  - **for** every process q **do**
    - send <heartbeat, p> to q
- **upon** <timer expires for q> **do**
  - suspected := suspected U {q}
  - initialize timer for process q using timeout[q]
- **upon** <heartbeat, q> **do**
  - if q in suspected then
    - suspected := suspected \ {q}
    - timeout[q] := timeout[q] + 1
  - initialize timer for process q using timeout[q]

Implementation

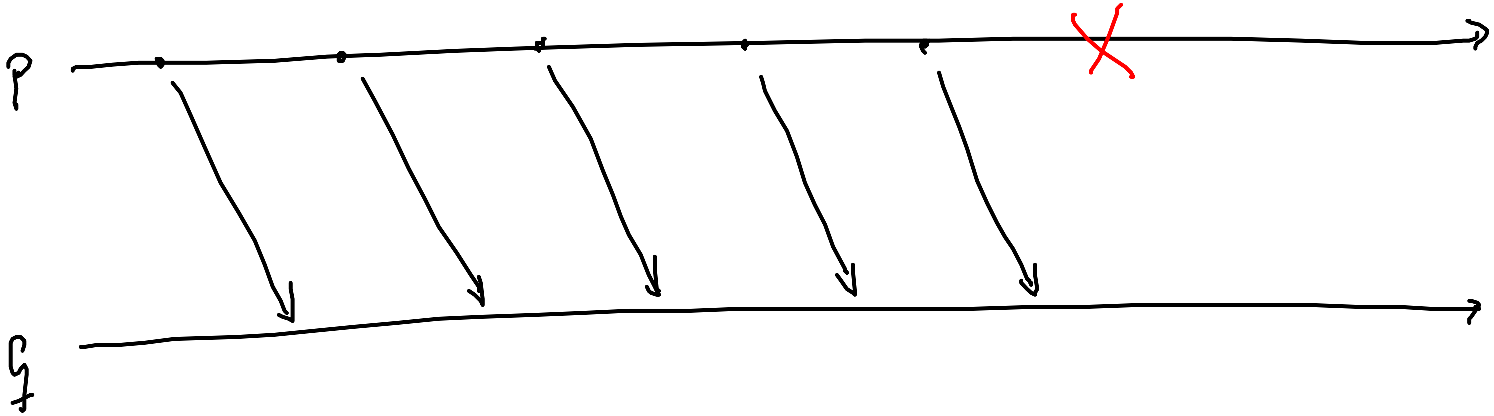


Correctness

- Under what timing assumptions does the failure detector implementation work?
  - synchronous,
  - partially synchronous,
  - asynchronous?

- Look at different cases (for two processes only):
  - (1) Synchronous, where initial timeout is accurate
  - (2) Synchronous with too small initial timeout
  - (3) Partially synchronous with proper timeout for synchronous phase
  - (4) Partially synchronous with too small timeout for synchronous phase
  - (5) Asynchronous

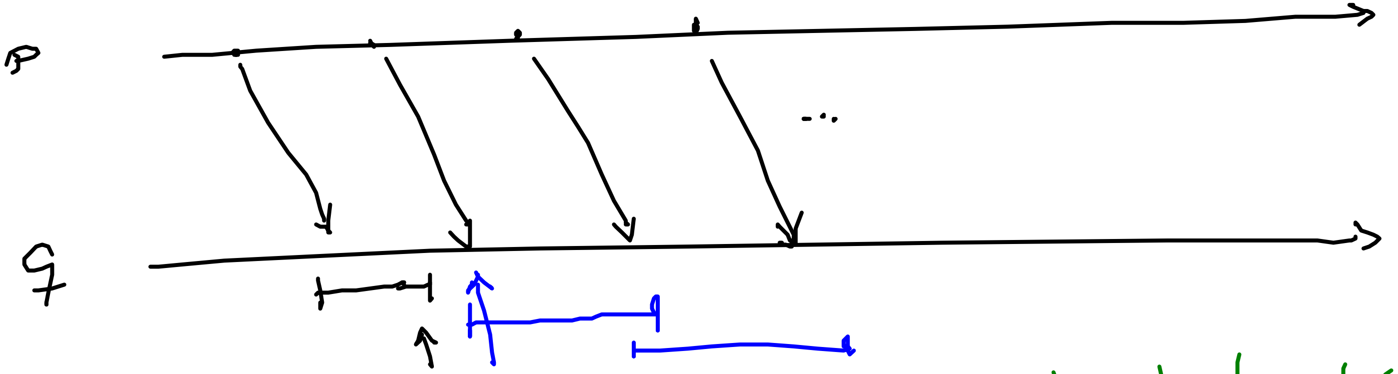
Case 1: Synchronous with proper timeout



strong completeness: if p crashes
⇒ p stops sending heartbeats
⇒ eventually timeout at q expires
⇒ q suspects p

strong accuracy: p is never suspected before it crashes
assume p is suspected by q
⇒ timeout at q has expired
⇒ no message received from p in timeout period
⇒ (synchronous system + proper timeout) p did not send heartbeat
⇒ p crashed

Case 2: Synchronous with improper timeout



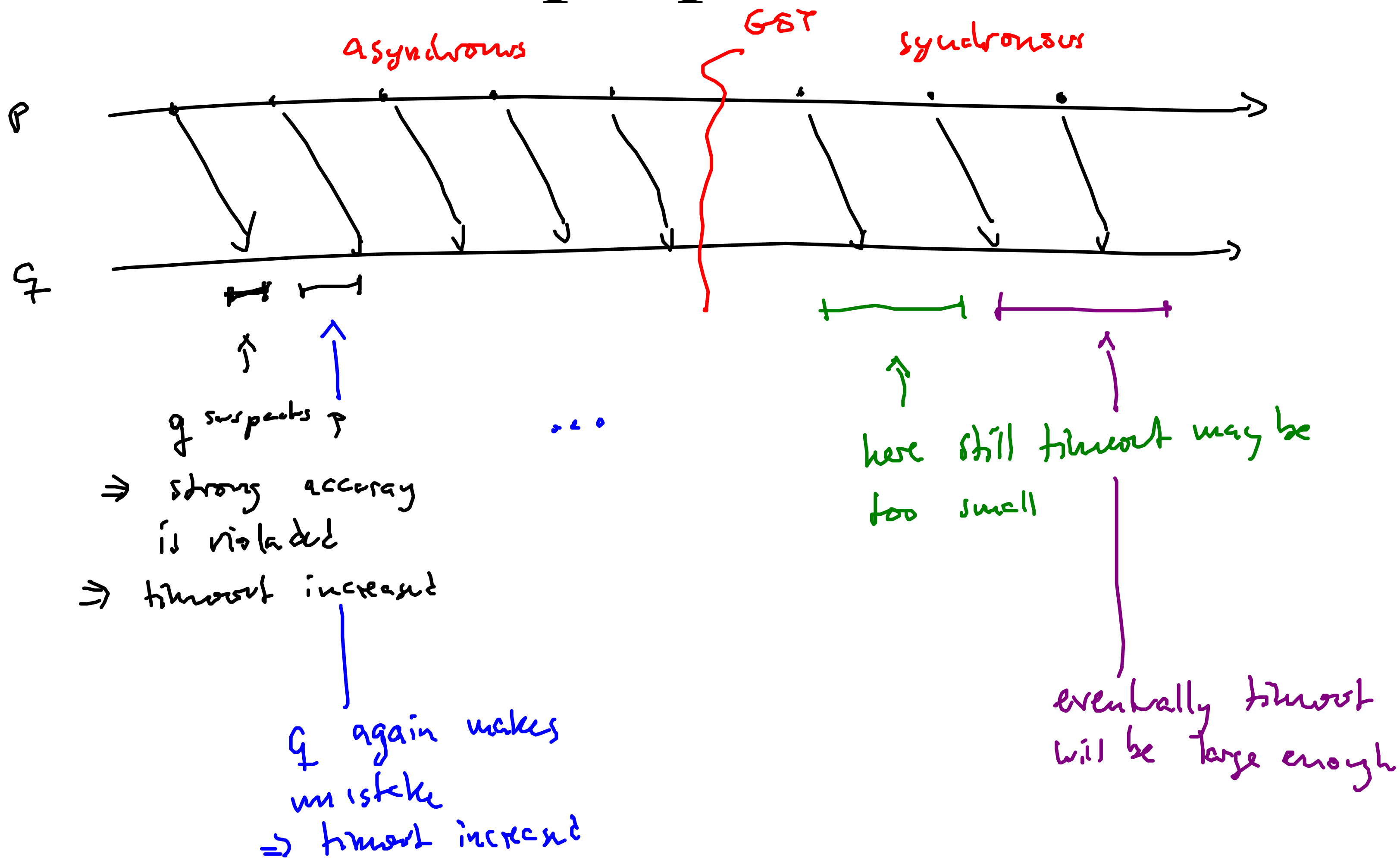
timeout runs out "too early"
⇒ q suspects p
⇒ violation of strong accuracy

next heartbeat arrives
⇒ increase timeout

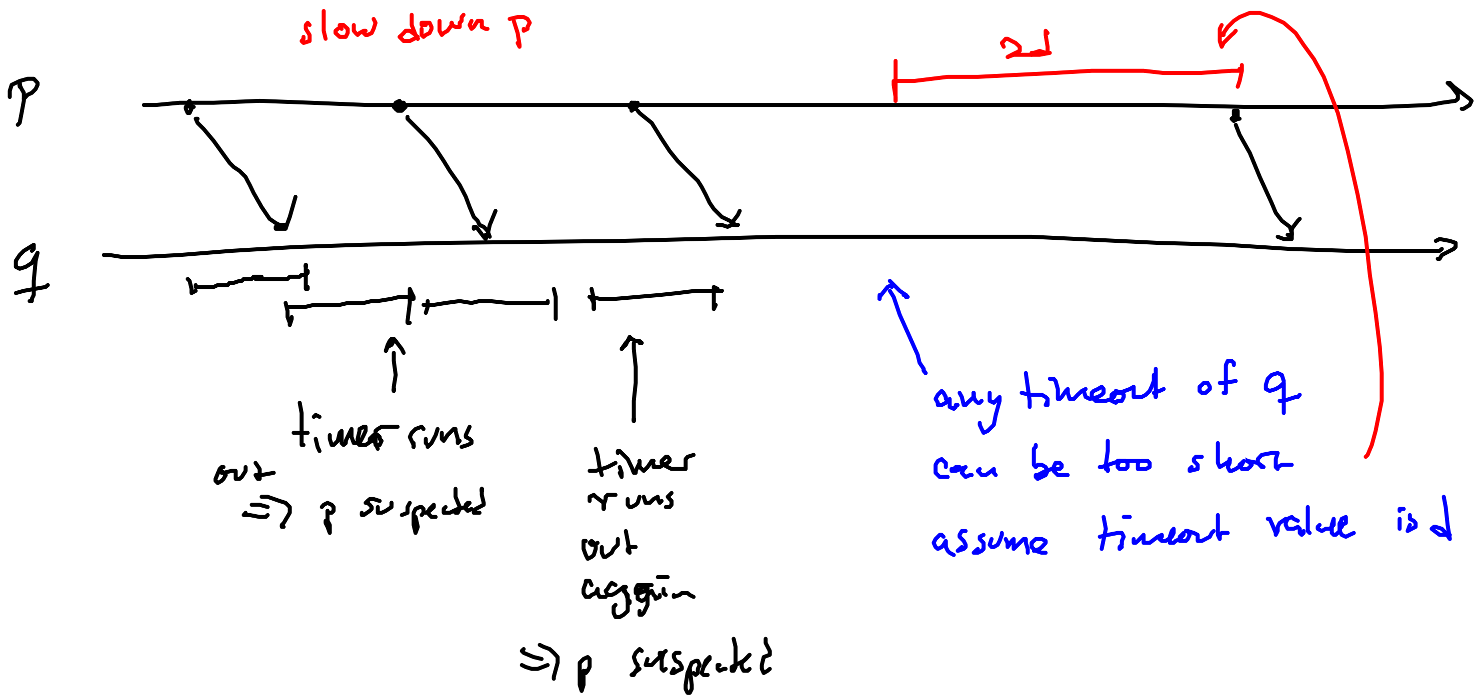
timeout gets larger with every mistake
⇒ eventually timeout is proper
⇒ eventually strong accuracy is satisfied

Case 3: Partially synchronous with proper timeout

Case 4: Partially synchronous with improper timeout



Case 5: Asynchronous

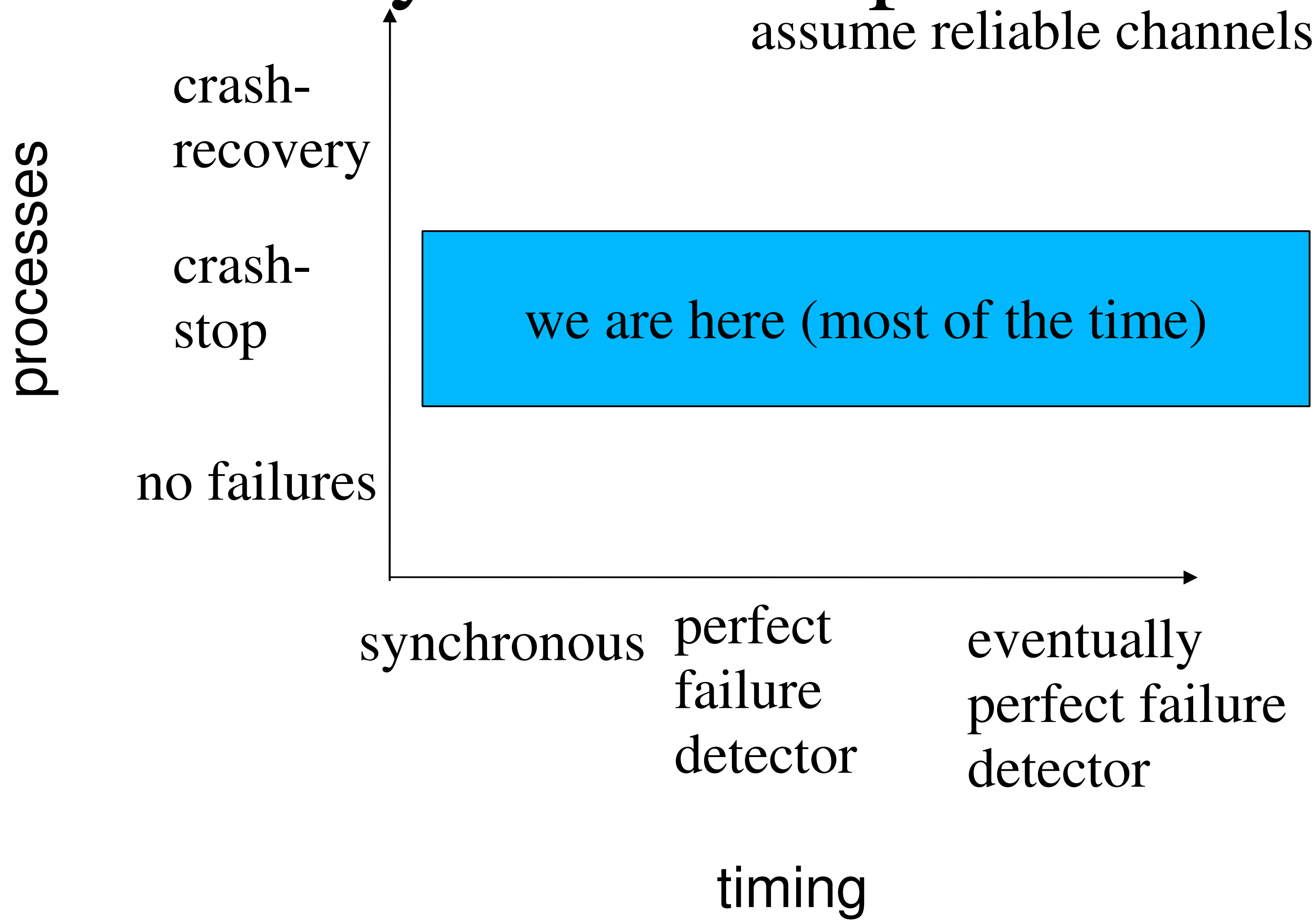


⇒ you can implement strong completeness
but you have no accuracy whatsoever!

Failure detection

- ***Perfect:***
  - *PFD1 (Strong Completeness):* Eventually, every process that crashes is permanently suspected by every correct process
  - *PFD2 (Strong Accuracy):* No process is suspected before it crashes
- ***Eventually Perfect:***
  - *PFD1*
  - *Eventual Strong Accuracy:* Eventually, no correct process is ever suspected

Summary of Assumptions



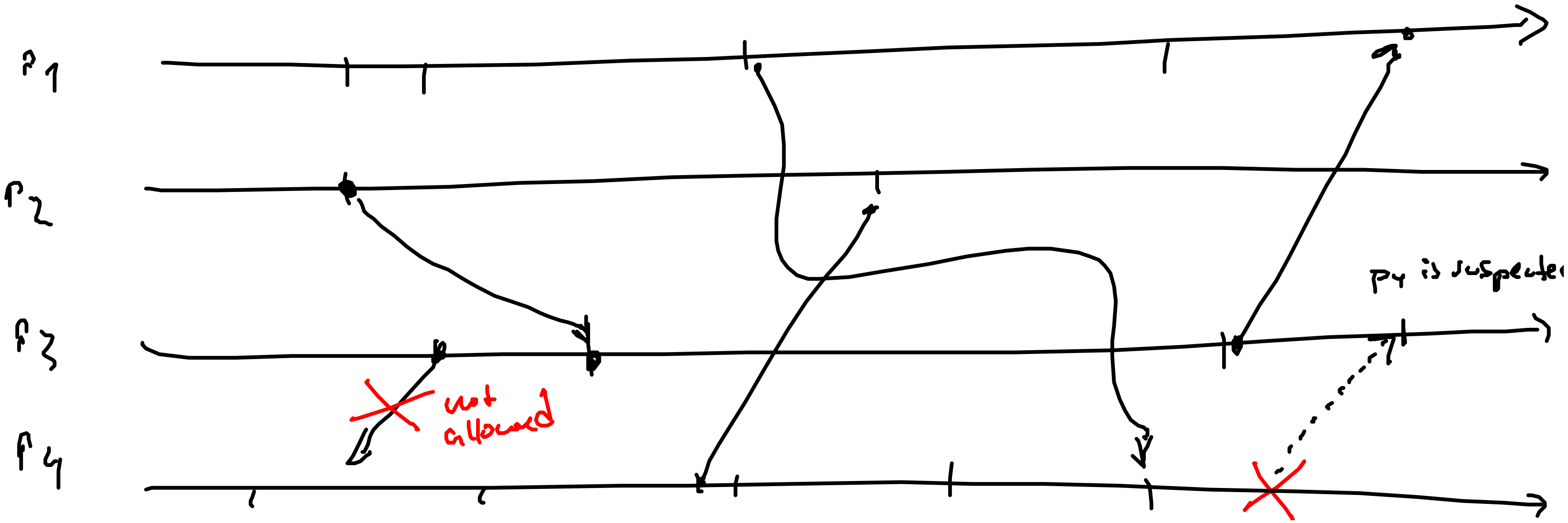
Overview

- Motivation:
  - Why study fault-tolerant distributed algorithms?
  - Where are they used?
- Formalizing distributed algorithms
  - Basic abstractions: processes and channels
  - The software stack
- How should we study fault-tolerant distributed algorithms?
  - Specifications: safety and liveness
  - Assumptions about processes, channels, and their failures
  - Example: building perfect communication links
  - Assumptions about timing
  - **Depicting algorithm runs**
- Distributed algorithms HOWTO (summary)

Algorithms

- Algorithms use events to communicate within a local stack of software layers
- Events have different types
- Algorithms "relate" indication events to request events
- We depict algorithms using space/time diagrams

Space/Time Diagram



Rules of Space/Time Diagrams

- Process execution goes from left to right
- Message arrows connect send and receive events at processes
- Message arrows must point to the right (may never point vertically or to the left)
- For perfect failure detectors: crash and suspicion can be interpreted as send and receive of a virtual message
  - Rules for messages hold analogously
  - Similarly rules hold for eventually perfect failure detectors which have “become perfect”

- Rubber-band transformations:
  - As long as rules above are satisfied, space/time diagrams can be stretched or squashed arbitrarily, resulting in legitimate space/time diagrams

Overview

- Motivation:
  - Why study fault-tolerant distributed algorithms?
  - Where are they used?
- Formalizing distributed algorithms
  - Basic abstractions: processes and channels
  - The software stack
- How should we study fault-tolerant distributed algorithms?
  - Specifications: safety and liveness
  - Assumptions about processes, channels, and their failures
  - Example: building perfect communication links
  - Assumptions about timing
  - Depicting algorithm runs
- **Distributed algorithms HOWTO (summary)**

Distributed algorithms HOWTO

- Make assumptions explicit:
  - Processes with crash-stop faults
  - Reliable channels, fully connected topology
  - Perfect failure detector at every process
- Define the problem:
  - Specify the interface operations (request, indication events)
  - Specify the safety and liveness properties of the problem based on the interface
- Design an algorithm:
  - Design software stack
  - Give local algorithms for each layer
- Study the algorithm:
  - Try to argue precisely for correctness
  - Use space/time diagrams to play with the algorithm

Summary

- Motivation:
  - Why study fault-tolerant distributed algorithms?
  - Where are they used?
- Formalizing distributed algorithms
  - Basic abstractions: processes and channels
  - The software stack
- How should we study fault-tolerant distributed algorithms?
  - Specifications: safety and liveness
  - Assumptions about processes, channels, and their failures
  - Example: building perfect communication links
  - Assumptions about timing
  - Depicting algorithm runs
- Distributed algorithms HOWTO (summary)

Coming next

- Study the problem of reliable broadcast in more detail
  - Assume crash-stop processes with reliable channels and a perfect failure detector
  - Specify reliable broadcast (different flavors)
  - Implement reliable broadcast (several algorithms)