DoC 437 - 2009

Distributed Algorithms

Prof. Alexander Wolf (alw)

assisted by
Tiejun Ma (tma)

With grateful acknowledgment to others, including:
Christos Karamanolis, Jeff Kramer, Jeff Magee, and Fernando Pedone

Intended course outcomes

- Appreciation for the challenges of designing algorithms for distributed systems

- Familiarity with several classical models, problems, and algorithms

    consensus, commit, logical time, reliable broadcast, routing

- Ability to reason about distributed algorithms

    correctness and performance

Some resources

- Distributed Systems. S. Mullender (ed.). Addison-Wesley, 1993
- Distributed Algorithms. N. Lynch. Morgan Kaufmann, 1996
- Introduction to Distributed Algorithms. G. Tel. Cambridge Univ. Press, 2000

DoC 437 - 2009

Distributed Algorithms

Part 1: Introduction

Systems and algorithms

- What is the difference between a *system* and an *algorithm*?

- Why do we tend to separate the study of algorithms from that of systems?

How do we study algorithms?

- *Experimentation:* implement and observe

  demonstrates the algorithm, subject to the setting in which it is constructed and executed

  even if we lack understanding of why it works, experience lets us reuse it in similar settings

- *Modeling and analysis:* abstract and reason

  provides a deep understanding, subject to the fidelity of the model

  even if we lack detail, we can predict properties

Models

- What is a model?

  an abstract representation of *properties*, *relationships*, and/or *behavior*

- 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

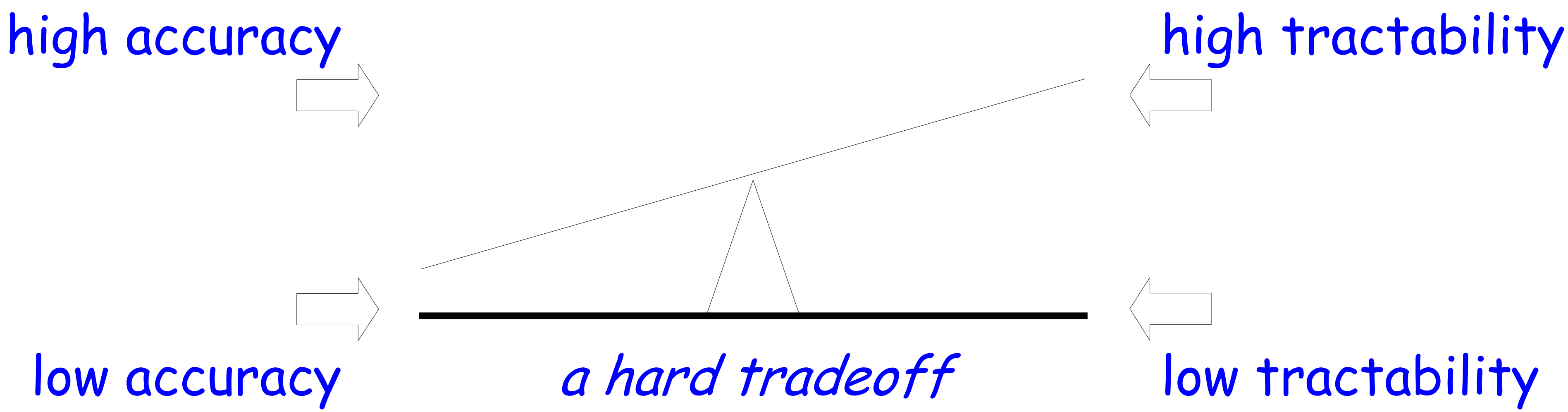
What characterizes a "good" model?

- Accurate

    yields true properties of the modeled object

- Tractable

    amenable to meaningful and practical analysis



What do we want from a model?

- To establish correctness

  does the algorithm result in a *desired outcome*? must be able to state the desired outcome within the model

- To predict performance

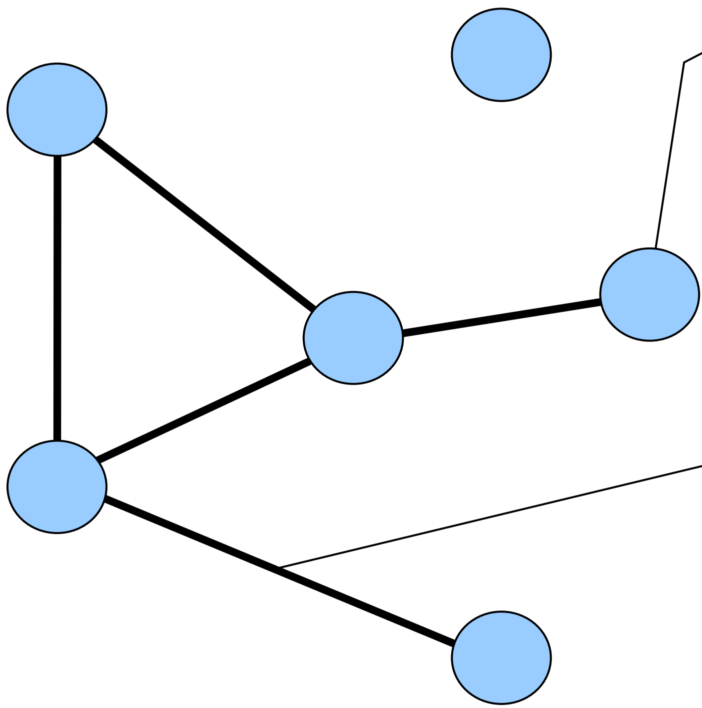
  what are the *costs* of using this algorithm?

  usually stated in terms of: state size, complexity, and amount of communication

- Are there other uses for models?

Basic modeling elements

components: processes and channels/links



process
- computational resource
- local state

channel / link
- communication resource
- message transport

A “distributed” algorithm?

- No shared global information

  processes make control decisions using *local state*

  processes deduce communication effects based on *messages* they send and receive

- No shared global time frame

  processes observe progress of computation through at best a *partial order of events*

- Nondeterministic behavior

  cannot predict the exact sequence of global states from study of the algorithm

Distribution and failure

- Distributed algorithms exhibit *partial failure*

  as the number of components increase…

  ✓ the likelihood that *all* components will fail *decreases*

  ✗ the likelihood that *some* components will fail *increases*

- Process failure

  infinitely slow or corrupt computation

- Communication loss

  a message may never arrive (infinite delay)

- Communication delay

  computations at processes progress while messages are in transit

A coordination problem

- The model

  processes *A* and *B* communicate by sending and receiving messages on a bidirectional channel

  A and *B* can execute two actions, α and β

  neither process can fail, but the channel may lose messages

- The desired outcome

  both processes take the same action, and neither takes both actions

- The outcome is *impossible* under the model

A coordination problem

impossibility proof

- Proof (by contradiction)

  any protocol executes in rounds of message exchanges: first *A* sends a message to *B*, then *B* sends a message to *A*, and so on

  let *P* be the protocol that solves the problem using the fewest rounds

  assume that the last message is sent by *A*, and let it be called *m*

A coordination problem

impossibility proof (cont.)

observation #1: the action taken by *A* cannot depend on *m*, because its receipt could never be learned by *A* (it is the last message)

observation #2: the action taken by *B* cannot depend on *m*, because *B* must make the same choice of action even if *m* is lost

since the action chosen by *A* and *B* does not depend on *m*, it follows that *m* is not needed and so we can construct a *P'* in which one fewer message is sent

but this is a contradiction, since *P* is then not using the fewest rounds ■

What is learned from this example?

- This simple model implies that...

  all protocols between two processes under this model are equivalent to a series of message exchanges

  all actions taken by a process under this model depend only on the sequence of messages it has received

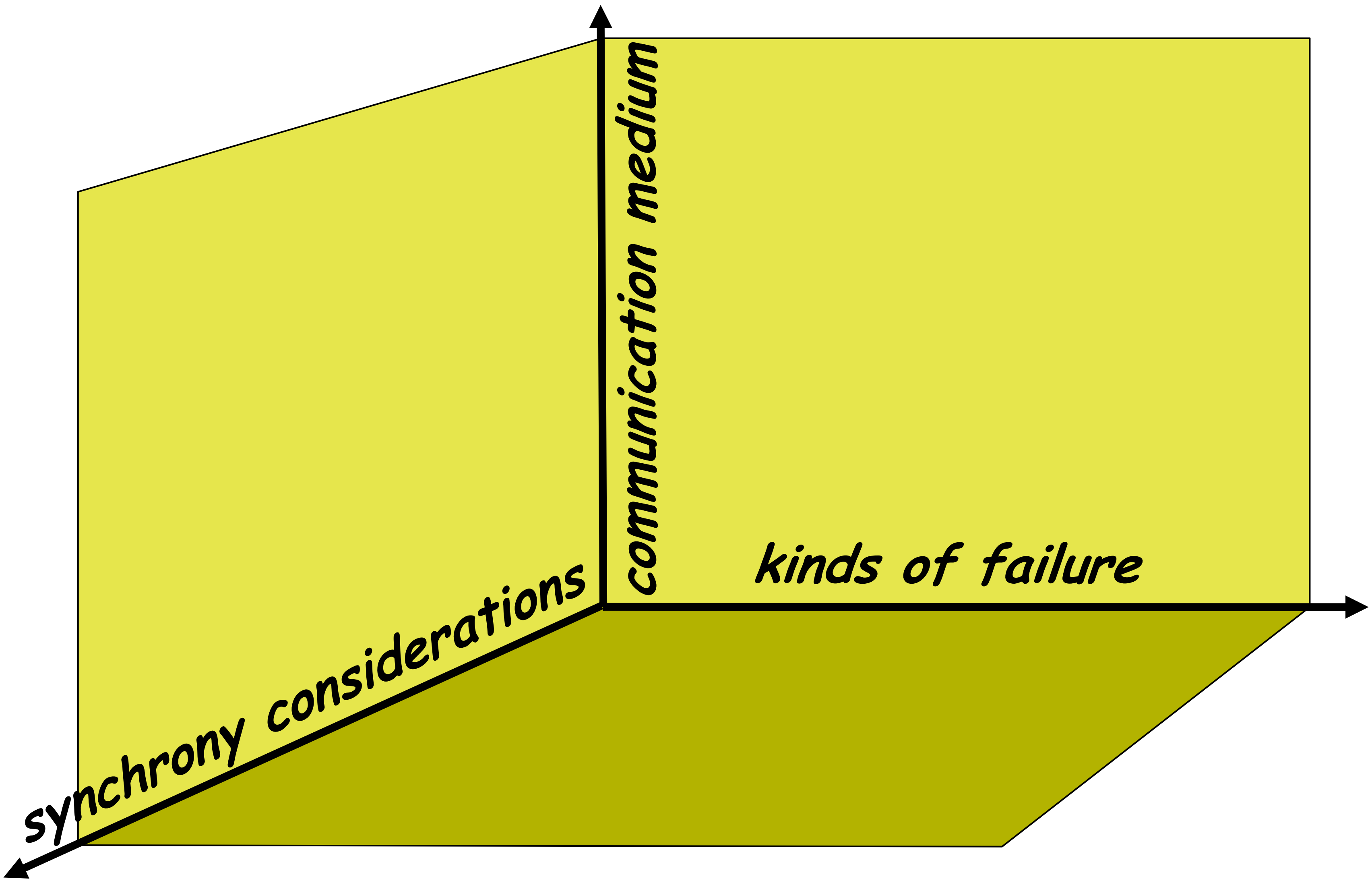
We could enrich the model

- What if channels never fail?
- What if channels only lose *k* messages?
- What if processes know when a message is lost?
- What about the cases for more than 2 processes?

- The choice of model is key to the design of the algorithm

  question to ask: what is the fidelity of the model?

Models of distributed computing



Communication medium

- communication medium
  - message-passing network
    - point-to-point address
    - content-based address
    - broadcast

      specific topologies

      ring, star, tree, ...
  - shared memory

Message-passing network

- Modeled as a graph

  nodes are processes, edges are channels
- Basic operations on channels

  send and receive
- Various semantics for channels

  blocking or non-blocking

  buffered or non-buffered

  bidirectional or unidirectional

Kinds of failure (i.e., failure models)

- Assign responsibility for faulty behavior to the system's components (processes and channels)

  if a message fails to get through, which component was responsible? sender? receiver? the channel?

    why is it important to know this?

  we are not interested in counting occurrences of faults (e.g., to compute mean time between failure)

  but, we are (sometimes) interested in understanding how many faulty components can be tolerated

    "t-fault tolerance"

Examples of failure models

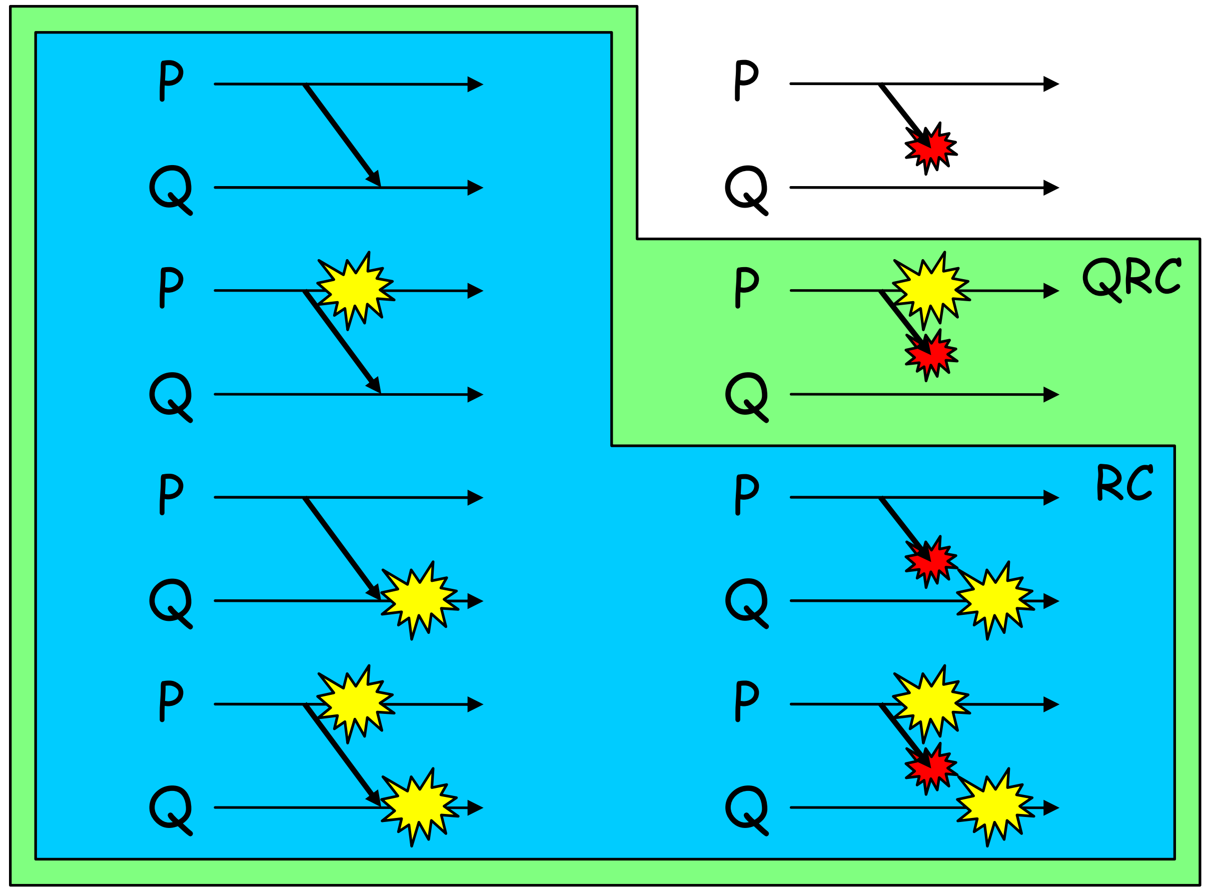
- *Failstop:* a process fails by halting; once it halts, the process remains in that state; other processes can detect the failed process
- *Crash:* a process fails by halting; once it halts, the process remains in that state; other processes may or may not detect the failed process
- *Crash+Link:* a process fails by halting; once it halts, the process remains in that state; a link fails by losing some messages, but does not delay, duplicate, or corrupt messages

More examples of failure models

- *Receive omission:* a process fails by receiving only a subset of the messages sent to it, or by halting and remaining halted
- *Send omission:* a process fails by sending only a subset of the messages it attempts to send, or by halting and remaining halted
- *General omission:* receive omission + send omission
- *Byzantine:* a process fails by exhibiting arbitrary behavior

Characterizing channels

- *Reliable channel:* if *P* sends a message *m* to *Q* and *Q* does not crash, then *Q* receives *m*
- *Quasi-reliable channel:* if *P* sends a message *m* to *Q* and both *P* and *Q* do not crash, then *Q* receives *m*
- *Unreliable channel:* *if… and … then …*



Properties of failure-free networks

- Process specifications

  if a process has not reached a final state, *eventually* it will execute another *step*

  Liveness

- Communication specifications

  process Q receives message m from process P *at most once* and only if P has *previously sent* m to Q

  Safety

  if P sends m to Q and Q takes *infinitely many steps*, then Q *eventually receives* m from P

  Liveness

- Benefit: can construct formal specification of the failure model

  check for failure is then check for *property violation*

Two approaches to fault tolerance

when bad things are likely to happen

- *Robust* algorithms

  assume correct processes always behave correctly

  never wait for all processes to complete

  tolerate failures by using replication and voting

  used when dealing with permanent failures

- *Stabilizing* algorithms

  assume correct processes eventually behave correctly

  can start in any state (possibly faulty), but will eventually behave correctly

  used when dealing with transient failures

Example: decision problems

the basics

- Robust algorithms typically try to solve some *decision* problem

  each correct process irreversibly "decides", usually represented as a specific value of a local variable

  special cases of decision: *consensus* and *election*

- Basic requirements on decision problems

  termination: all correct processes eventually decide

  consistency: constraint on decision outcome

  for consensus, all decide the same

  for election, all decide that same one is different

Example: decision problems

distributed decision makers need to communicate

- Reliable broadcast

  all correct processes deliver the same set of messages

  the set only contains messages from correct processes

- Atomic (reliable) broadcast

  A reliable broadcast where it is guaranteed that every process receives its messages in the same order as all the other processes

Example: decision problems

atomic broadcast → consensus

- Let every node broadcast either 0 or 1

  decide on the first number that is received

  since every correct process will receive the messages in the same order, they will all decide on the same value

- Solving reliable atomic broadcast is therefore equivalent to solving consensus

  how do we do that?

A fundamental result

how many failures can a robust algorithm tolerate?

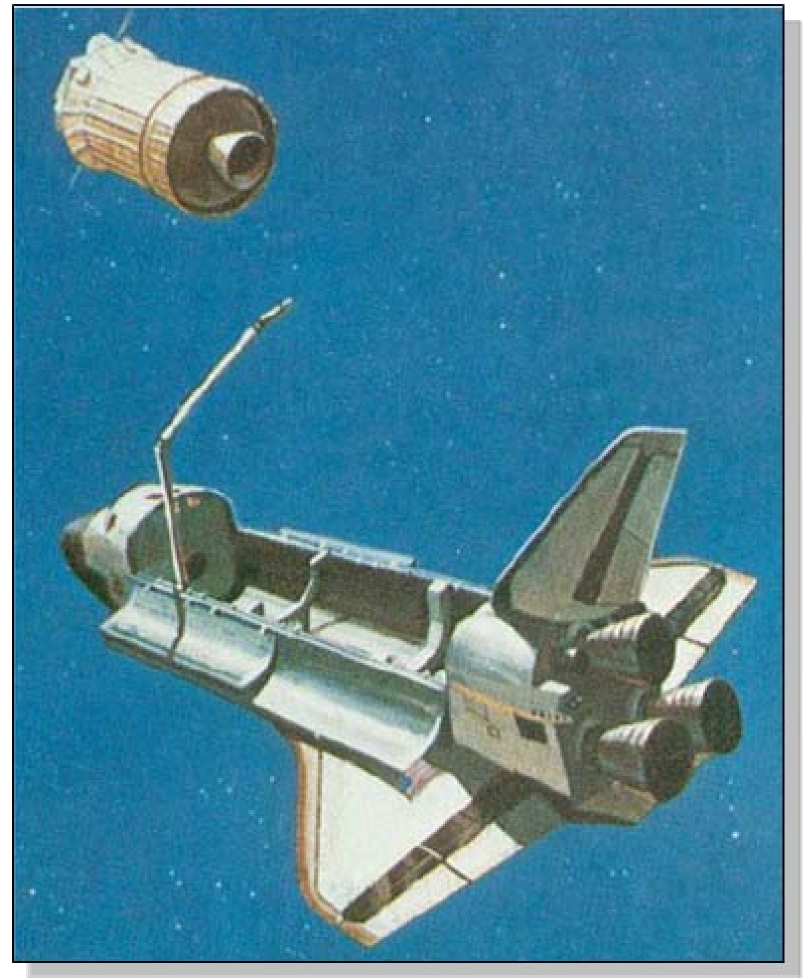
- Given N+1 processes, a (correct) robust algorithm should be able to tolerate

    N/2 benign failures

    N/3 Byzantine failures

- Example: PASS (Primary Avionics Software System) developed for the Space Shuttle by IBM in 1981

    five computers, four running identical software and one running equivalent software



Synchrony considerations

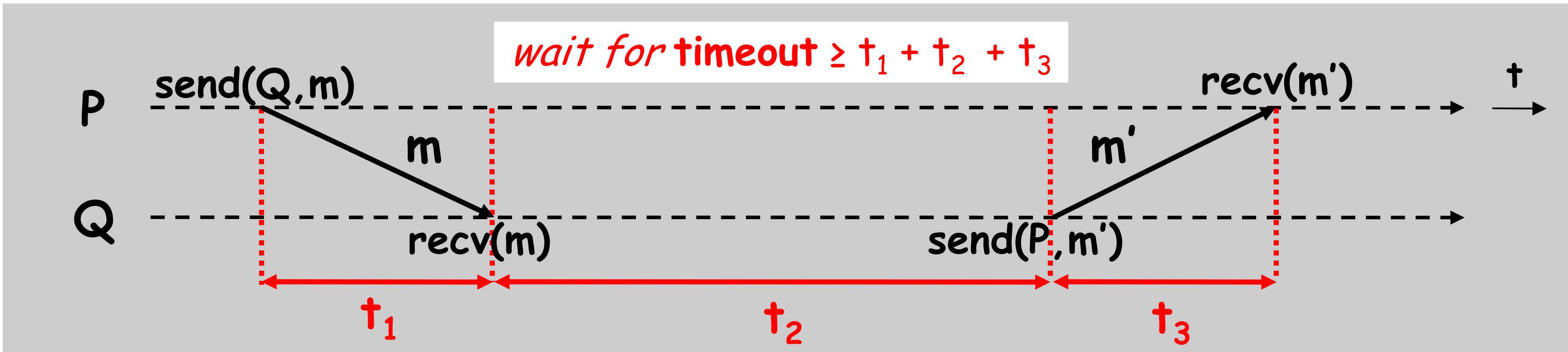
synchronous network model

- Known upper bound on time required for a process to execute a computation step
- Known upper bound on message delay
- Processes have perfectly synchronized physical clocks

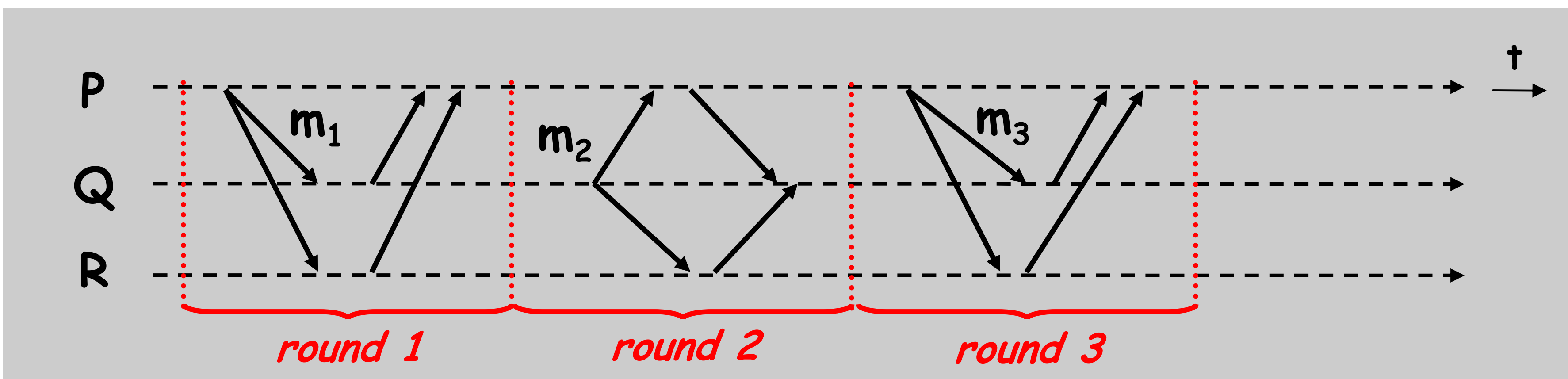
in practice, implementations assume a known bounded *drift factor* (a.k.a., “approximately synchronized”)

Consequences of synchronous model

- Can use *timeouts* to detect failures



- Can organize computation into *rounds*



Synchrony considerations

asynchronous network model

- No bound on time required for a process to execute a computation step

  however, time is finite

- No bound on message delay
- Processes do not have perfectly synchronized (or even approximately synchronized) physical clocks

Consequences of asynchronous model

- Most general model of synchrony

    an algorithm that works under the asynchronous model necessarily will work under the synchronous model

    why not design all algorithms using the asynchronous model?

    why not design all algorithms using the synchronous model?

Synchronous vs. asynchronous

example: leader election problem

- A set of processes P_1, P_2, …, P_n must select a leader
- Each P_i has a unique identifier UID(i)
- Processes start at the same time and communicate using broadcast

Synchronous vs. asynchronous

asynchronous solution to leader election problem

- Each P_i broadcasts (i, UID(i)), and waits for the receipt of every broadcast message

- P_i elects the process with the smallest UID

Synchronous vs. asynchronous

synchronous solution to leader election problem

- Let τ be a constant > max(delay)

  for simplicity, assume local computation takes no time

- Each process P_i waits until either

  P_i receives a broadcast

  or

  τ * UID(i) time units elapse on P_i's clock, at which time P_i broadcasts UID(i)

- The first process to broadcast is elected

Summary

- To describe a distributed algorithm, you must first specify...

    communication model

    process and channel failure models

    assumptions on the number (usually maximum) of process or channel failures

    synchrony model