Specification and Implementation of Distributed Systems

Seif Haridi – Royal Institute of Technology
Peter Van Roy – Université catholique de Louvain
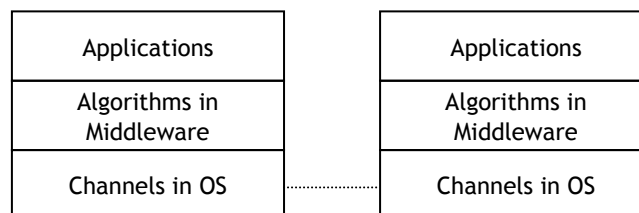haridi(at)kth.se
peter.vanroy(at)uclouvain.be

9/27/21   1

■1

Need for Distributed Abstractions

- Core of any distributed system is a set of distributed algorithms
  - Implemented as a middleware between network (OS) and the application
- Reliable applications need underlying services stronger than network protocols (eg TCP, UDP)

Applications		Applications
Algorithms in Middleware		Algorithms in Middleware
Channels in OS	···········	Channels in OS

9/27/21   2

■2

1

Need for Distributed Abstractions (2)

- ■ Network protocols aren't enough
  - ❑ Communication
    - ■ Reliability guarantees (eg TCP) only offered for one-to-one communication (client-server)
    - ■ How to do group communication?

  - ❑ High-level services
    - ■ Sometimes one-to-many communication isn't enough
    - ■ Need reliable high-level services

Abstractions in this course

reliable broadcast
causal order broadcast
total order broadcast
terminating reliable broadcast

shared memory
consensus
atomic commit
group membership

■3

Reliable Distributed Abstractions

- ■ Example 1: *reliable broadcast*
  - ❑ Ensure that a message sent to a group of nodes is received (delivered) by all or none

- ■ Example 2: *atomic commit*
  - ❑ Ensure that the nodes reach the same decision on whether to commit or abort a transaction

■4

Event-based Component Model

■5

The Model

- Each node models a sequential program
- Assume a global clock
- At each tick, one node takes one step
  - Computation Step: local computation
  - Communication Step: deliver a message
- Different models
  - Receive 1 msg and send 1 msg [Guerraoui]
  - At most receive 1 msg, and send at most 1 msg to each neighbor [Lynch]
  - Receive k msgs, and send at most 1 msg to each neighbor [Attiya & Welch]
- We saw Attiya & Welch, but the others are very similar

■6

Event-based Programming

- Nodes execute programs:

  - Each program consists of a set of modules or component specifications.
  - At runtime these are deployed as components
  - The components form a software stack
  - Components interact via events (with attributes):

  ```
  upon event <ReceiveEvent, attr1, attr2,...> do
          // local computation (single-threaded)
          …
          trigger <SendEvent, attr3, attr4,...>
  ```

Event-based Programming

- Events give different kinds of information
  - Messages (most of the time)
  - Timers
  - Boolean conditions (e.g. x==5 & y<9)

- Three event categories
  - Requests (flow downward)
  - Indications (the responses, flow upward)
  - Confirmations (special type of indication, like an OK or ACK)

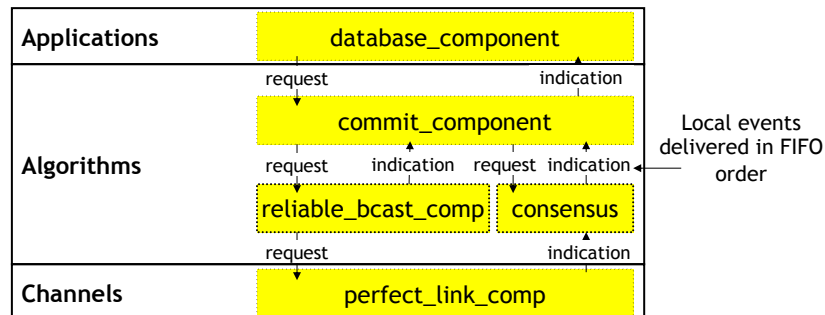# Modules on one Node

- Stack of modules on a single node
- Requests flow down, indications flow up

Applications	database_component
Algorithms	commit_component / reliable_bcast_comp / consensus
Channels	perfect_link_comp

request   indication

request   indication   request   indication

Local events delivered in FIFO order

request   indication

■9

Send and Receive Instructions

- Send instruction is called *trigger*:
  - ❑ Send a request or an indication
  - ❑ Example: Send a message with data

    trigger *<sendBcast | dest*, [data1, data2, …] >

- Receive instruction is called *upon event*:
  - ❑ Receive a request or an indication
  - ❑ Example: Receive a message with data
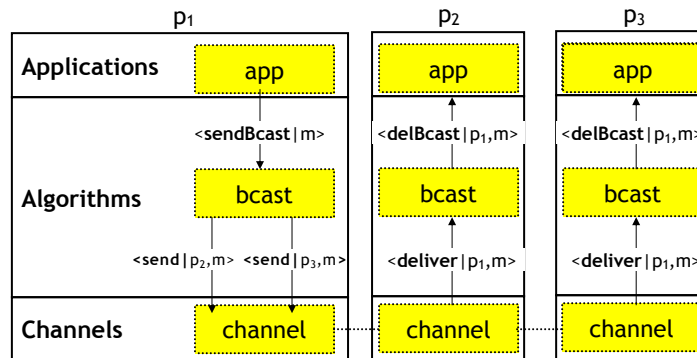
    upon event *<delBcast | src*, [data1,data2, …]> **do**

■10

Modules on several Nodes

- This application uses a Broadcast component
  - Which uses a channel module to broadcast

	p_1	p_2	p_3
Applications	app	app	app
	$<$**sendBcast**$|m>$	$<$**delBcast**$|p_1,m>$	$<$**delBcast**$|p_1,m>$
Algorithms	bcast	bcast	bcast
	$<$**send**$|p_2,m>$ $<$**send**$|p_3,m>$	$<$**deliver**$|p_1,m>$	$<$**deliver**$|p_1,m>$
Channels	channel	channel	channel

■11

Specification

■12

Specification of a Service

- How to specify a distributed service?

  - Interface (a.k.a. Contract, API)
    - Requests
    - Responses

  - Correctness Properties
    - Safety
    - Liveness

  - Model
    - Failure assumptions
    - Synchrony assumptions

  declarative
  specification
  "what"
  a.k.a. problem

  - Implementation
    - Composed of other services
    - Adheres to interface and satisfies correctness
    - Has internal events

  imperative,
  many possible
  "how"

■13

Simple Interface Example (1)

Module 1.1 Interface of a printing module

Module:

　　Name: Print.

Events:

　　Request: ⟨ *PrintRequest* | rqid, str ⟩: Requests a string to be printed. The token rqid is an identifier of the request.

　　Confirmation:⟨ *PrintConfirm* | rqid ⟩: Used to confirm that the printing request with identifier rqid succeeded.

■14

Simple Interface Example (2)

Algorithm 1.1 Printing service

Implements:
    Print.

upon event $\langle$ *PrintRequest* | rqid, str $\rangle$ **do**
    print str;
    trigger $\langle$ *PrintConfirm* | rqid $\rangle$;

■15

Simple Interface Example (3)

Module 1.2 Interface of a bounded printing module

Module:

    Name: BoundedPrint.

Events:

    Request: $\langle$ *BoundedPrintRequest* | rqid, str $\rangle$: Request a string to be printed. The token rqid is an identifier of the request.

    Confirmation:$\langle$ *PrintStatus* | rqid, status $\rangle$: Used to return the outcome of the printing request: Ok or Nok.

    Indication:$\langle$ *PrintAlarm* $\rangle$: Used to indicate that the threshold was reached.

■16

Simple Interface Example (4)

Algorithm 1.2 Bounded printer based on (unbounded) printing service

Implements:
    BoundedPrint.

Uses:
    Print.

upon event $\langle$ *Init* $\rangle$ **do**   *<Init>* automatically
    bound := PredefinedThreshold;   generated upon
                                           component creation

upon event $\langle$ *BoundedPrintRequest* | rqid, str $\rangle$ **do**
    if bound > 0 **then**
        bound := bound-1;
        trigger $\langle$ *PrintRequest* | rqid, str $\rangle$;
        if bound = 0 **then trigger** $\langle$ *PrintAlarm* $\rangle$;
    else
        trigger $\langle$ *PrintStatus* | rqid, Nok $\rangle$;

upon event $\langle$ *PrintConfirm* | rqid $\rangle$ **do**
    trigger $\langle$ *PrintStatus* | rqid, Ok $\rangle$;

■17

Safety and Liveness

- Correctness is always expressed in terms of
  - Safety and liveness

- Safety
  - Properties that state that "something bad never happens"

- Liveness
  - Properties that state that "something good eventually happens"

■18

Correctness Example (1)

- Correctness of You in LSINF2345
  - Safety
    - You should never fail the exam

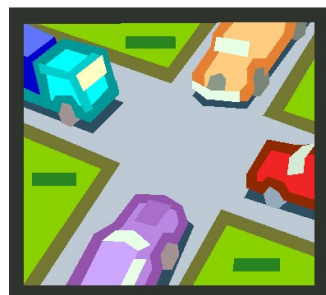 - Liveness
    - You should eventually take the exam

19

Correctness Example (2)

- Correctness of traffic lights at intersection
  - Safety
    - More than one direction should never have a green light at same time

  - Liveness
    - Every direction should eventually get a green light

20

Safety & Liveness All That Matters

- A property P is a function that takes an execution (!) and returns true/false
  - I.e., P is a predicate

- "Any [property] can be expressed as the conjunction of a safety property and a liveness property"
  - Alpern & Schneider, Inf.Proc.Letters 1985

21

Prefixes and Extensions

- The prefix of execution E is the first *k* (for some *k>0*) configurations of E, with events
  - I.e., cut off the tail of E
  - I.e., finite beginning of E

- An extension of a prefix P is any execution that has P as a prefix
  - The extension continues P

22

Safety Formally Defined

- Informally, property P is a safety property if
  - Every execution E violating P "goes bad", i.e., it has a bad event s.t. every execution starting like E and behaving like E up to the bad event (including), will violate P regardless of what it does afterwards
  - When an execution "goes bad", it "stays bad"!

- Formally, a property P is a safety property if
  - Given any execution E such that P(E)=false,
  - There exists a prefix of E s.t. every extension of that prefix gives an execution F s.t. P(F)=false

Safety Example

- Point-to-point message communication
  - Safety P:
    - A message sent is delivered at most once

  - Take an execution where a message is delivered more than once
    - Cut off the tail after the second delivery
    - Any extension will give an execution which also violates the required property

Liveness Formally Defined

- A property P is a liveness property if
  - Given any execution E such that P(E)=true,
  - For every prefix F of E, there exists an extension of F s.t. P(F)=true

- "As long as there is life there is hope"

Liveness Example

- Point-to-point message communication
  - Liveness P:
    - A message sent is delivered at least once

  - Take the prefix of any execution
    - If prefix contains delivery, any extension satisfies P
    - If prefix doesn't contain the delivery, extend it so that it contains a delivery, then the prefix + extended part will satisfy P

Formal Definitions Visually

Execution E

∃ prefix
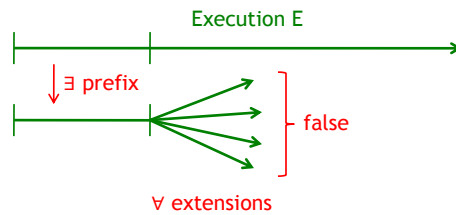
∀ extensions — false

- Safety can always be made false in finite time
- **Safety is false** for an execution E then there exists a prefix such that all extensions are false (system is broken)

Execution E

∀ prefixes
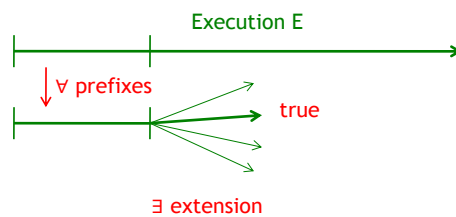
true

∃ extension

- Liveness can only be made false in infinite time
- **Liveness is true** for an execution E then for all prefixes there exists an extension that is true (system can progress)

■27

More on Safety

- Safety can only be
  - ❑ satisfied in infinite time (you're never safe)
  - ❑ violated in finite time (when the bad happens)

- Often involves the word "never", "at most", "cannot",…

- Sometimes called "partial correctness"

■28

More on Liveness

- Liveness can only be
  - satisfied in finite time (when the good happens)
  - violated in infinite time (there's always hope)

- Often involves the words "eventually", "must", "at least"
  - Eventually means at some (often unknown) point in "future"

- Liveness is often just "termination"

29

Pondering Safety and Liveness

- Why not define safety to be a predicate that is true in every configuration in an execution? [d]

- Is every property really either liveness or safety?
  - Every message should be delivered exactly 1 time [d]

30

Failure Assumptions for Nodes

31

Model/Assumptions

- Specification needs to specify the model
  - Assumptions needed for the algorithm to be correct

- Model includes assumptions on
  - Failure behavior of processes & channels
  - Timing behavior of processes & channels

32

Node Failures

- Nodes may fail in four ways:
  - Crash-stop
  - Omission
  - Crash-recovery
  - Byzantine/Arbitrary

- Nodes that don't fail in an execution are correct

33

Crash-stop Failures

- Crash-stop failure
  - Node stops taking steps
    - Not sending msgs
    - Nor receiving msgs

- Default failure model is crash-stop
  - Hence, do not recover
  - But are nodes really not allowed to recover? [d]

34

Omission Failures

- Node omits sending or receiving messages
  - Some differentiate between
    - Send omission
      - Not sending msgs node has to send according to its algo
        - Formally, an event removing element from outbuf[i]

    - Receive omission
      - Not receiving messages that have been sent to node
        - Formally, an event removing element from inbuf[i]

  - For us, omission failure covers both types…

Crash-recovery Failures

- The node might crash
  - It stops taking steps, not receiving and sending messages

- It may recover after crashing
  - Special <Recovery> event automatically generated
  - Formal model: Restarting in some initial recovery state

- Has access to stable storage
  - May read/write (expensive) to permanent storage device
  - Storage survives crashes
  - E.g., save state to storage, crash, recover, read saved state…

Crash-recovery Failures (2)

- Failure different in crash-recovery model

  - A node is faulty in an execution if
    - It crashes and never recovers, or
    - It crashes and recovers infinitely often (unstable)

  - Hence, a correct node may crash and recover
    - As long as it is a finite number of times

  - A correct node will eventually (after finite time) do no more crashes

37

Byzantine Failures

- Byzantine/Arbitrary failures
  - A node may behave arbitrarily
    - Sending messages not specified by its algorithm
    - Updating its state not specified by its algorithm

  - May behave maliciously, attacking the system
    - Several malicious nodes might collude (work together to break the system)

38

Fault-tolerance Hierarchy

- There is a hierarchy among the failure types
    - Which one is a special case of which? [d]

- Crash-stop is a special case of Omission
    - Omission restricted to omitting everything after a certain event

39

Fault-tolerance Hierarchy (2)

- Assume a special case of crash-recovery
    - Where nodes use only stable storage (no other memory)
- In this case crash-recovery is identical to omission
    - Crashing, recovering, and reading last state from storage
    - Just same as omitting send/recv while being crashed

- Assume a special case of crash-recovery
    - Where nodes use some volatile memory
        - Then recovered nodes might not be able to restore all of state
    - Thus crash-recovery extends omission with amnesia

- Therefore Omission is a special case of Crash-recovery
    - Omission restricted to omitting everything after a certain event
        - Possibly recovering, not allowing for amnesia
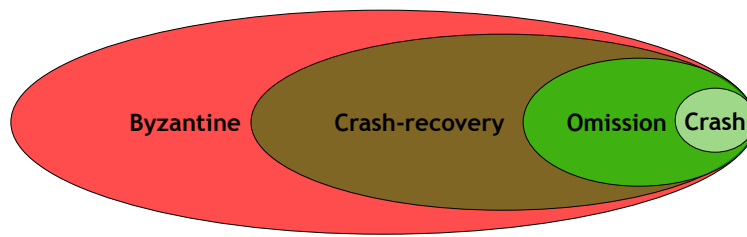
40

Fault-tolerance Hierarchy (3)

- Crash-recovery is a special case of Byzantine
  - Since Byzantine allows anything

- Byzantine tolerance $\rightarrow$ crash-recovery tol.
  - Crash-recovery $\rightarrow$ omission, omission $\rightarrow$ crash

■41

Failure Assumptions for Channels

■42

Channel failure modes

- **Fair-Loss Links**
  - Channel delivers any message sent with non-zero probability (no network partitions)

- **Stubborn Links**
  - Channel delivers any message sent infinitely many times

- **Perfect Links**
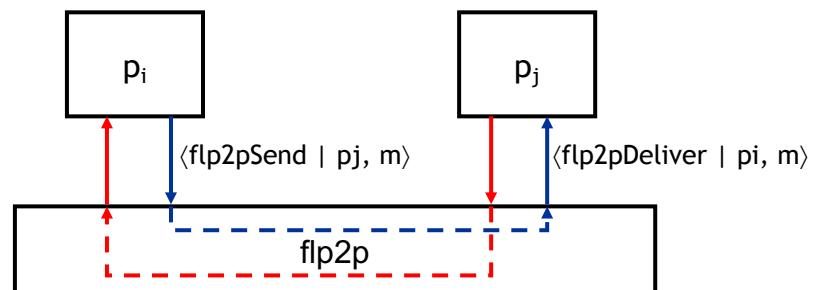 - Channel that delivers any message sent exactly once

■43

Fair Loss Links

p_i p_j

$\langle$flp2pSend | pj, m$\rangle$ $\langle$flp2pDeliver | pi, m$\rangle$

flp2p

■44

Fair-loss links: Interfaces

- **Module:**
  - Name: FairLossPointToPoint (flp2p)

- **Events:**
  - **Request**: ⟨flp2pSend | dest, m⟩
    - Request transmission of message m to node dest

  - **Indication**:⟨flp2pDeliver | src, m⟩
    - Deliver message m sent by node src

- **Properties:**
  - *FL1, FL2, FL3.*

■45

Fair-loss links

- Properties
  - *FL1. Fair-loss*: If m is sent infinitely often by p_i to p_j, and neither crash, then m is delivered infinitely often by p_j

  - *FL2. Finite duplication:* If a m is sent a finite number of times by p_i to p_j, then it is delivered a finite number of times by p_j
    - I.e., a message cannot be duplicated infinitely many times

  - *FL3. No creation:* No message is delivered unless it was sent

■46

Stubborn links: interface

- **Module:**
  - Name: StubbornPointToPoint (sp2p)

- **Events:**
  - **Request**: ⟨sp2pSend | dest, m⟩
    - Request the transmission of message m to node dest
  - **Indication:**⟨sp2pDeliver src, m⟩
    - deliver message m sent by node src

- **Properties:**
  - *SL1, SL2*

■47

Stubborn links

- Properties
  - *SL1. Stubborn delivery*: if a node p_i sends a message m to a correct node p_j, and p_i does not crash, then p_j delivers m an infinite number of times

  - *SL2. No creation:* if a message m is delivered by some node p_j, then m was previously sent by some node p_i

■48

Implementing Stubborn Links

- Implementation
  - Use the Lossy link (fair-loss link)
  - Sender stores every message it sends in **sent**
  - It periodically resends all messages in **sent**

- Correctness
  - *SL1. Stubborn delivery*
    - If node doesn't crash, it will send every message infinitely many times. Messages will be delivered infinitely many times. Lossy link may only drop a (large) fraction.
  - *SL2. No creation*
    - Guaranteed by the Lossy link

Stubborn Links in the Formal Model

- **Consider an execution**
  $E = <c_0, e_1, c_1, e_2, c_2,...>$

- *Stubborn property*
  - If message m appears in outbuf[p] for link i↔j of a node i in E
    - Then del'(i,j,m) will happen infinitely often, without removing m from outbuf[p]

- *Finite duplication* and *No creation*
  - Similarly

Algorithm (sl)

- **Implements:** StubbornLinks (sp2p)
- **Uses:** FairLossLinks (flp2p)

- **upon event** $\langle$Init$\rangle$ **do**
  - sent := $\varnothing$
  - startTimer(TimeDelay)

- **upon event** $\langle$Timeout$\rangle$ **do**
  - **forall** (dest, m) $\in$ sent **do**
    - **trigger** $\langle$flp2pSend | dest, m$\rangle$
  - startTimer(TimeDelay)

■51

Algorithm (sl) (2)

- **upon event** $\langle$sp2pSend | dest, m$\rangle$ **do**
  - sent := sent $\cup$ { (dest, m) }

- **upon event** $\langle$flp2pDeliver | src, m$\rangle$ **do**
  - **trigger** $\langle$sp2pDeliver src, m$\rangle$

■52

Perfect links: interface

- **Module:**
  - Name: PerfectPointToPoint (pp2p)

- **Events:**
  - **Request:** ⟨pp2pSend | dest, m⟩
    - Request the transmission of message m to node dest
  - **Indication:** ⟨pp2pDeliver | src, m⟩
    - deliver message m sent by node src

- **Properties:**
  - *PL1, PL2, PL3*

■53

Perfect links aka Reliable links

- *Properties*
  - *PL1. Reliable Delivery*: If neither p_i nor p_j crashes, then every message sent by p_i to p_j is eventually delivered by p_j

  - *PL2. No duplication:* Every message is delivered at most once

  - *PL3. No creation:* No message is delivered unless it was sent

■54

Perfect links aka Reliable links

- Which one is safety/liveness/neither?

- *PL1. Reliable Delivery*: If neither p_i nor p_j crashes, then every message sent by p_i to p_j is eventually delivered by p_j (liveness)

- *PL2. No duplication:* Every message is delivered at most once (safety)

- *PL3. No creation:* No message is delivered unless it was sent (safety)

■55

Perfect Link Implementation

- Implementation
  - Use Stubborn links
  - Receiver keeps log of all received messages in **Delivered**
    - Only deliver (pp2pDeliver) messages that weren't delivered before

- Correctness
  - *PL1. Reliable Delivery*
    - Guaranteed by Stubborn link. In fact the Stubborn link will deliver it infinite number of times

  - *PL2. No duplication*
    - Guaranteed by our log mechanism

  - *PL3. No creation*
    - Guaranteed by Stubborn link (and its lossy link? [D])

■56

Algorithm (pl)

- **Implements:** PerfectLinks (pp2p).
- **Uses:** StubbornLinks (sp2p).

- **upon event** ⟨Init⟩ **do** delivered := ∅

- **upon event** ⟨pp2pSend | dest, m⟩ **do**
  - **trigger** ⟨sp2pSend dest, m⟩

- **upon event** ⟨sp2pDeliver | src, m⟩ **do**
  - **if** m ∉ delivered **then**
    - delivered := delivered ∪ { m }
    - **trigger** ⟨pp2pDeliver | src, m⟩

■57

Default Assumptions in Course

- We assume perfect links (aka reliable) most of the course (unless specified otherwise)

- Roughly, reliable links ensure messages exchanged between correct nodes are delivered exactly once

- NB. Messages are uniquely identified and
  - the message identifier includes the sender's identifier
  - i.e. if "same" message sent twice, it's considered as two different messages

- Stubborn links used mostly in the crash-recovery

■58

TCP vs perfect links

- How does TCP efficiently maintain **delivered** log? [d]

- Use sequence numbers
    - ACK: "I have received everything up to byte X"

■59

Synchrony Assumptions

■60

Synchrony Assumptions

- **Synchrony assumptions** relate to
  - different processing speeds of nodes
  - different speeds of messages (channels)

- Three basic types of systems:
  - Asynchronous system
  - Synchronous system
  - Partially synchronous system

61

Asynchronous Systems

- No timing assumption on nodes and channels
  - Processing time varies arbitrarily
  - No bound on transmission time

- We know it well from our formal model!
  - Causality
    - Lamport clocks (or vector clocks) to observe causality

  - Total order not observable, no access to clock
    - We used Computation Theorem

- Is Internet asynchronous? [d]

62

Synchronous Systems

- Model assumes
  - Synchronous computation
    - Known upper bound on node processing delays

  - Synchronous communication
    - Known upper bound on message transmission delay

  - Synchronous physical clocks
    - Nodes have local physical clock
    - Known upper bound clock drift and clock skew

- Why study synchronous systems? [d]

- 63

Partial Synchrony

- Asynchronous system
  - Which eventually becomes synchronous
    - Cannot know when, but in every execution, some bounds eventually will hold

- It's just a way to formalize the following
  - *Your algorithm will have a long enough time window, where everything behaves nicely (synchrony), so that it can achieve its goal*
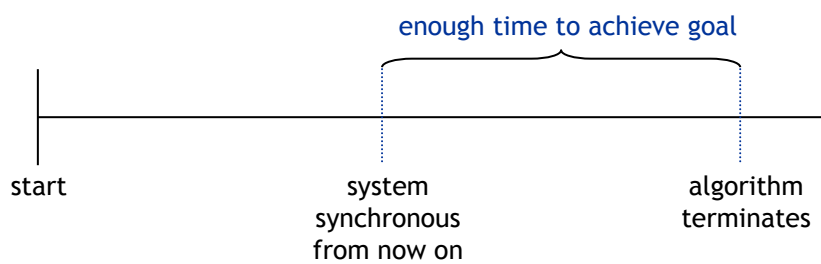
- Are there such systems? [d]

- 64

Partial Synchrony (2)

- *Your algorithm will have a long enough time window, where everything behaves nicely (synchrony), so that it can achieve its goal*
  - Mostly useful for terminating algorithms

- 65