Introduction to Basic Abstractions

Seif Haridi

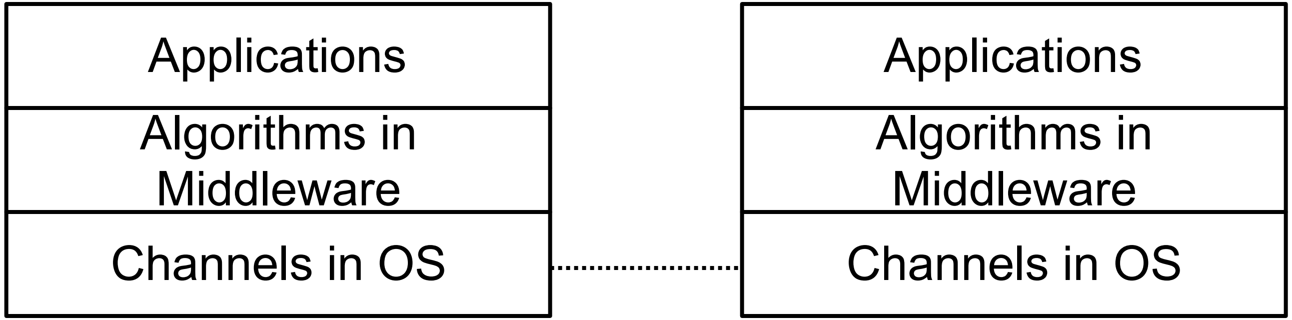
haridi@kth.se

Need of 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 (e.g. TCP, UDP)

Need of Distributed Abstractions

- Core of any distributed system is a set of distributed algorithms
    - Implemented as a middleware between network (OS) and the application


Applications

Algorithms in Middleware

Channels in OS

Applications

Algorithms in Middleware

Channels in OS

Need of Distributed Abstractions

- Network protocols aren’t enough
  - Communication
    - Reliability guarantees (e.g. TCP) only offered for one-to-one communication (client-server)
    - How to do group communication?

Abstractions in this course

Reliable broadcast
Causal order broadcast
Total order broadcast

Need of Distributed Abstractions

- Network protocols aren’t enough
  - High-level services
    - Sometimes many-to-many communication isn’t enough
    - Need reliable high-level services

Abstractions in this course

Shared memory
Consensus
Atomic commit
Replicated state machine

Reliable distributed abstractions

- Example 1: ***reliable broadcast***
  - Ensure that a message sent to a group of processes is received (delivered) by all or none
- Example 2: ***atomic commit***
  - Ensure that the processes reach the same decision on whether to commit or abort a transaction

Event-based Component Model

Distributed Computing Model

- Set of processes and a network (communication links)
- Each process runs a local algorithm (program)
- Each process makes computation steps

- The network makes computation steps
  - to store a message sent by a process
  - to deliver a message to a process

- Message delivery triggers a computation step at the receiving process

The Distributed Computing Model

- Computation step at a process
  - Receives a message (external, input)
  - Performs local computation
  - Sends one or more messages to some other processes (external, output)

- Communication step:
  - Depends on the network abstraction
  - Receives a message from a process, or
  - Delivers a message to a process

Inside a Process

- A process consists of a set of components (automata)
- Components are concurrent
- Each component receives messages through an input FIFO buffer
- Sends messages to other components
- Events are messages between components in the same process
- Events are handled by procedures (actions) called Event Handlers

Inside a Process

Event-based Programming

- Process executes program
  - Each program consists of a set of modules or component specifications
  - At runtime these are deployed as components
  - The components in general form a software stack

Event-based Programming

- Process executes program
  - Components interact via events (with attributes):
  - Handled by Event Handlers

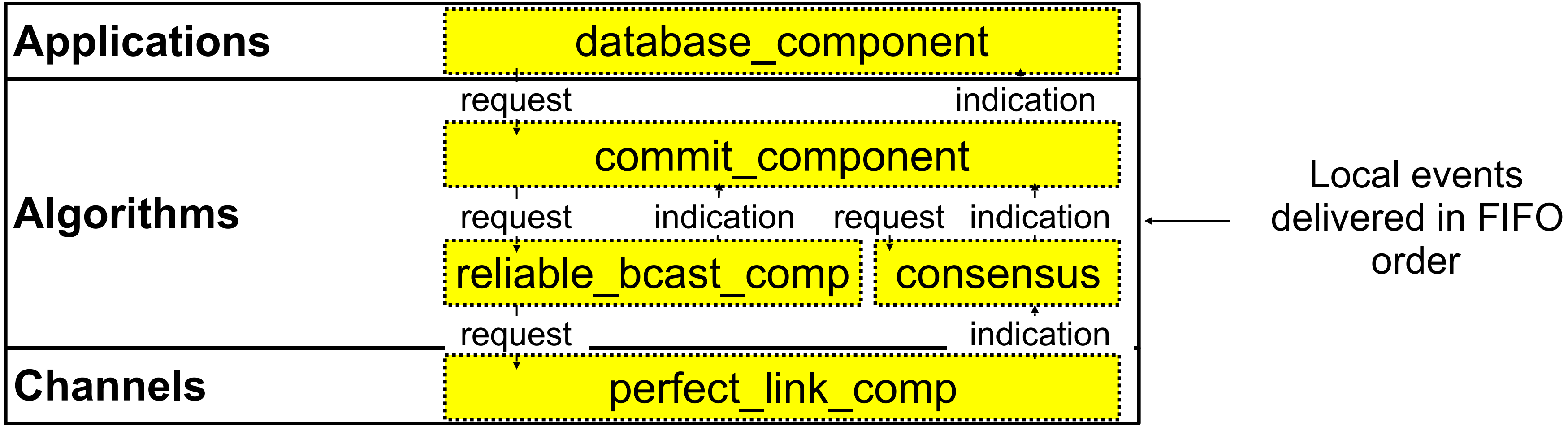
```
on event <co_i Event_1, attr1, attr2,...> do
          // local computation
          trigger <co_j Event_2, attr3, attr4,...>
```

Event-based Programming

- Events can be almost anything
  - Messages (most of the time)
  - Timers (internal event)
  - Conditions (e.g. x==5 & y<9)
- Two types of events
  - Requests
    - (flows downward) Inputs
  - Indications
    - (like responses/acks flows upward) Outputs

Components in a Process

- Stack of components in a single process



Channels as Modules

- Channels represented by modules (too)
    - Request event:
        - Send to destination some message (with data)

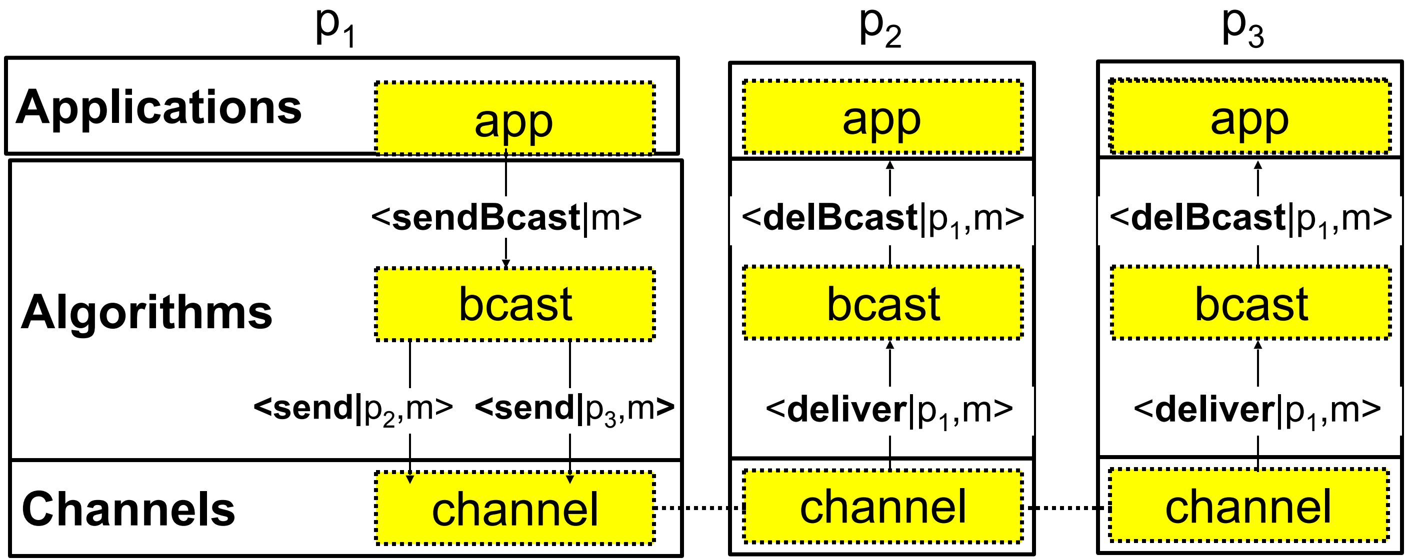
trigger <*send* | *dest*, [data1, data2, …] >

- Indication event:
    - Deliver from source some message (with data)

upon event <*deliver* | *src,* [data1,data2, …]> **do**

Example

- Application uses a Broadcast component
  - which uses channel component to broadcast



Specification

Specification of a Service

- How to specify a distributed service (abstract)?
  - Interface (aka Contract, API)
    - Requests
    - Responses
  - Correctness Properties
    - Safety
    - Liveness
  - Model
    - Assumptions on failures
    - Assumptions on timing (amount of synchrony)

declarative
specification
“what”
aka problem

……

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

imperative,
many possible
“how”

Simple Example: Job Handler

- **Module:**
  - **Name: JobHandler, instance *jh***
- **Events:**
  - **Request:** *⟨jh, Submit | job⟩ : Requests a job to be processed*
  - **Indication:** *⟨jh, Confirm | job⟩ : Confirms that the given job has been (or will be) processed*
- **Properties:**
  - *Guaranteed response: Every submitted job is eventually confirmed*

Implementation Example

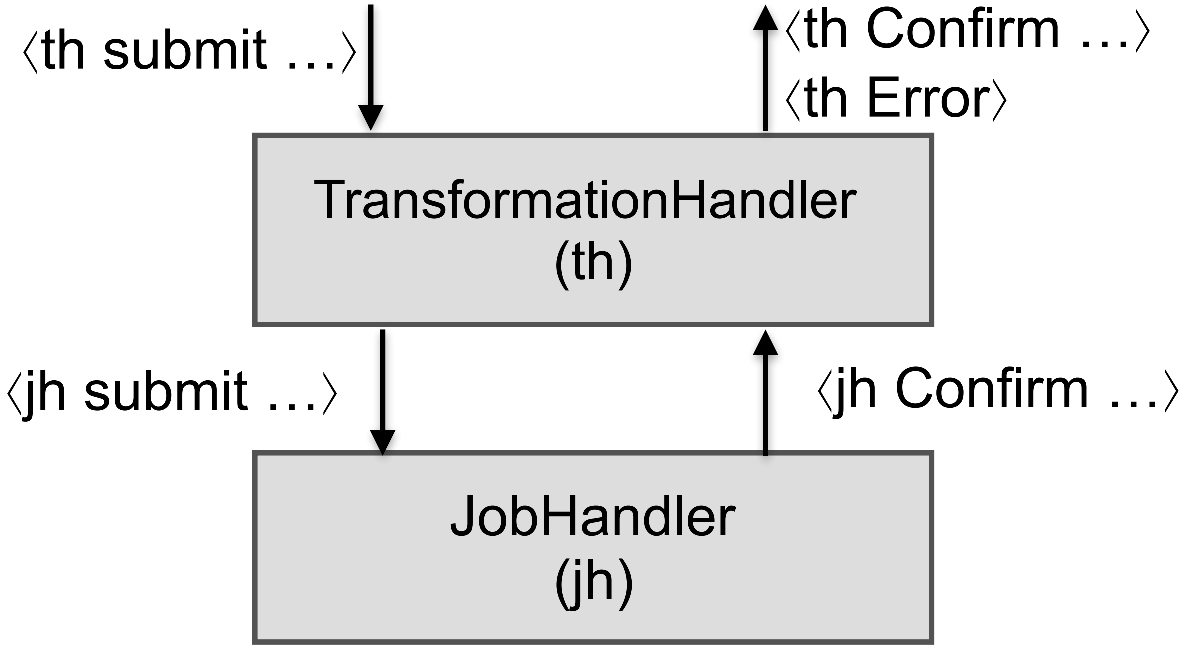
- Synchronous Job Handler
- **Implements:**
    - JobHandler, **instance *jh***
- **upon event** ⟨*jh, Submit | job*⟩ ***do***
    - process(*job)*
    - **trigger** ⟨*jh, Confirm | job*⟩

Another implementation: Asynchronous Job Handler

- **Implements:**
  - JobHandler, **instance *jh***
- **upon event** ⟨*jh, Init*⟩ ***do***
  - *buffer* := ∅
- **upon event ⟨*jh, Submit* | *job*⟩ *do***
  - *buffer* := *buffer* ∪ {*job*}
  - **trigger ⟨*jh, Confirm* | *job*⟩**
- **upon** *buffer* ≠ ∅ **do**
  - *job* := *selectjob* (*buffer*)
  - process(*job*)
  - *buffer* := *buffer* \ {*job*}

⟨..***Init***⟩ automatically generated upon component creation

Component Composition



⟨th submit …⟩

⟨th Confirm …⟩

⟨th Error⟩

TransformationHandler
(th)

⟨jh submit …⟩

⟨jh Confirm …⟩

JobHandler
(jh)

Properties
Safety and Liveness

Specification of a Service

- How to specify a distributed service (abstract)?
  - Interface (aka Contract, API)
    - Requests
    - Responses
  - Correctness Properties
    - Safety
    - Liveness
  - Model
    - Assumptions on failures
    - Assumptions on timing (amount of synchrony)

declarative
specification
“what”
aka problem

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

imperative,
many possible
“how”

Correctness

- Always expressed in terms of
  - Safety and liveness
- Safety
  - Properties that state that nothing bad ever happens
- Liveness
  - Properties that state that something good eventually happens

Correctness Example

- Correctness of You in ID2203x
  - Safety
    - You should never fail the exam (marking exams costs money)
  - Liveness
    - You should eventually take the exam (university gets money when you pass)

Correctness Example (2)

- Correctness of traffic lights at intersection
  - Safety
    - Only one direction should have a green light
  - Liveness
    - Every direction should eventually get a green light

Execution and Traces (reminder)

- An execution fragment of A is sequence of alternating states and events
  - $s_0, \ \varepsilon_1, s_1, \varepsilon_2, \ldots, s_r, \varepsilon_r, \ldots$
  - $(s_k, \varepsilon_{k+1}, s_{k+1})$ transition of A for $k \geq 0$
- An execution is execution fragment where s_0 is an initial state
- A trace of an execution E, trace(E)
  - The subsequence of E consisting of all external events
  - $\varepsilon_1, \varepsilon_2, \ldots, \varepsilon_r, \ldots$

Safety & Liveness All That Matters

- A trace property P is a function that takes a trace and returns true/false
  - i.e. P is a predicate

- Any trace property can be expressed as the conjunction of a safety property and a liveness property”

Safety Formally Defined

- The prefix of an trace T is the first *k* (for *k* ≥ 0) events of T
  - I.e. cut off the tail of T
  - I.e. finite beginning of T

- An extension of a prefix P is any trace that has P as a prefix

Safety Defined

- Informally, property P is a safety property if
  - Every trace T violating P has a bad event, s.t. every execution starting like T and behaving like T up to the bad event (including), will violate P regardless of what it does afterwards

Safety Defined

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

Safety Example

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

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 continuation (extension) will give an execution which also violates the required property

Liveness Formally Defined

- A property P is a liveness property if
  - Given any prefix F of an execution E,
  - There exists an extension of trace(F) for which P is 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

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, the prefix + extended part will satisfy P

More on Safety

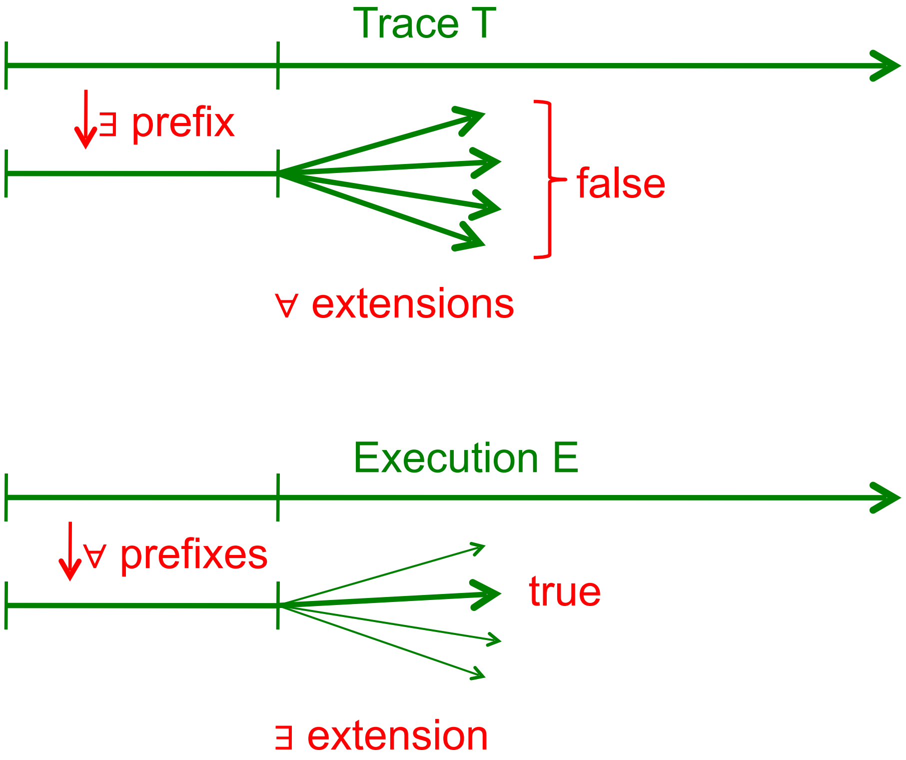
- Safety can only be
  - satisfied in infinite time (you’re never safe)
  - violated in finite time (when the bad happens)

- Often involves the word “never”, “at most”, “cannot”,…
- Sometimes called “partial correctness”

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, or must
  - Eventually means at some (often unknown) point in "future"
- Liveness is often just "termination"

Formal Definitions Visually



- Safety can always be made false in finite time
- Safety is false for an execution E if there exists a prefix such that all extensions are false
- Liveness can always be made true in finite time
- Liveness is true for an execution E if for all prefixes there exists an extension that is true

Pondering Safety and Liveness

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

- Every message is delivered at most once and
- Every message is delivered at least once

Process Failure Model

Specification of a Service

- How to specify a distributed service (abstract)?
  - Interface (aka Contract, API)
    - Requests
    - Responses
  - Correctness Properties
    - Safety
    - Liveness
  - Model
    - Assumptions on failures
    - Assumptions on timing (amount of synchrony)

declarative
specification
"what"
aka problem

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

imperative,
many possible
"how"

Model/Assumptions

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

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

Process failures

- Processes may fail in four ways:
  - Crash-stop
  - Omissions
  - Crash-recovery
  - Byzantine/Arbitrary
- Processes that don’t fail in an execution are correct

Crash-stop failures

- Crash-stop failure
  - Process stops taking steps
    - Not sending messages
    - Nor receiving messages

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

Omission failures

- Process omits sending or receiving messages
  - Some differentiate between
    - Send omission
      - Not sending messages the process has to send according to its algorithm
    - Receive omission
      - Not receiving messages that have been sent to the process
  - For us, omission failure covers both types

Crash-recovery Failures

- The process might crash
  - It stops taking steps, not receiving and sending messages
- It may recover after crashing
  - Special <Recovery> event automatically generated
  - 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

- Failure is different in crash-recovery model
  - A process is faulty in an execution if
    - It crashes and never recovers, or
    - It crashes and recovers infinitely often (unstable)
  - Hence, a correct process may crash and recover
    - As long as it is a finite number of time

Byzantine failures

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

  - May behave maliciously, attacking the system
    - Several malicious processes might collude

Fault-tolerance Hierarchy

Fault-tolerance Hierarchy

- Is there a hierarchy among the failure types
  - Which one is a special case of which? [d]
  - An algorithm that works correctly under a general form of failure, works correctly under a special form of failure

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

Fault-tolerance Hierarchy

- In Crash-recovery
  - Under assumption that processes use stable storage as their main memory

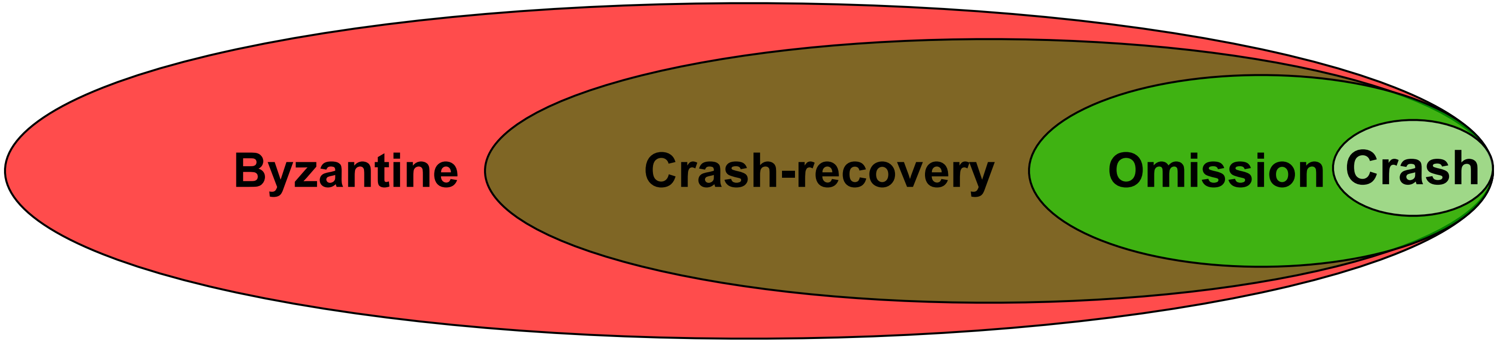
- Crash-recovery is identical to omission
  - Crashing, recovering, and reading last state from storage
  - Just same as omitting send/receiving while being crashed

Fault-tolerance Hierarchy

- In crash-recovery it is possible to use volatile memory
  - Then recovered nodes might not be able to restore all of state
  - Thus crash-recovery extends omission with amnesia
- Omission is special case of Crash-recovery
    - Crash-recovery , not allowing for amnesia

Fault-tolerance Hierarchy

- Crash-recovery special case of Byzantine
  - Since Byzantine allows anything
- Byzantine tolerance → crash-recovery tolerance
  - Crash-recovery → omission, omission → crash-stop



Channel Behavior (failures)

Specification of a Service

- How to specify a distributed service (abstract)?
  - Interface (aka Contract, API)
    - Requests
    - Responses
  - Correctness Properties
    - Safety
    - Liveness
  - Model
    - Assumptions on failures
    - Assumptions on timing (amount of synchrony)

declarative
specification
"what"
aka problem

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

imperative,
many possible
"how"

Channel failure modes

- Fair-Loss Links
  - Channels delivers any message sent with non-zero probability (no network partitions)
- Stubborn Links
  - Channels delivers any message sent infinitely many times
- Perfect Links
  - Channels that delivers any message sent exactly once

Channel failure modes

- Logged Perfect Links
  - Channels delivers any message into a receiver’s persistent store (message log)

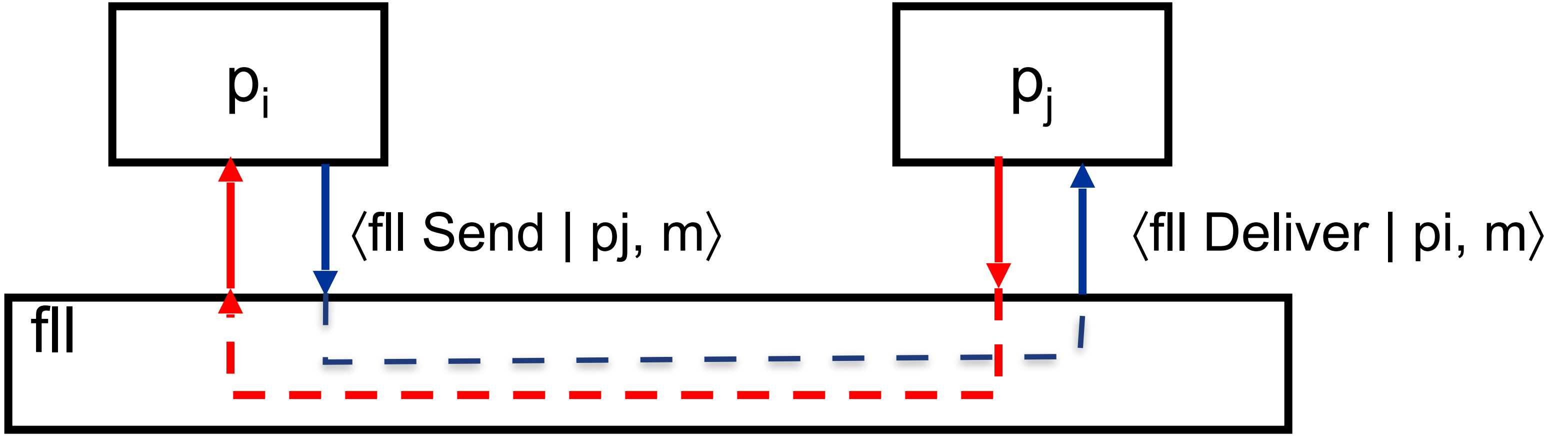
- Authenticated Perfect Links
  - Channels delivers any message m sent from process p to process q, that guarantees the m is actually sent from p to q

Fair Loss Links

Channel failure modes

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

Fair Loss Links (fll)



⟨fll Send | pj, m⟩

⟨fll Deliver | pi, m⟩

Fair-loss links: Interfaces

- **Module:**
  - Name: FairLossPointToPointLink **instance** fll
- **Events:**
  - **Request**: ⟨fll, Send | dest, m⟩
    - Request transmission of message m to process dest
  - **Indication**:⟨fll, Deliver | src, m⟩
    - Deliver message m sent by process src

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

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 at most 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

Stubborn Link

Channel failure modes

- Stubborn Links
  - Channels delivers any message sent infinitely many times

Stubborn links: interface

- **Module:**
  - Name: StubbornPointToPointLink **instance** sl
- **Events:**
  - **Request**: ⟨sl, Send | dest, m⟩
    - Request the transmission of message m to process dest
  - **Indication**:⟨sl, Deliver src, m⟩
    - deliver message m sent by process src
- **Properties:**
  - ***SL1, SL2***

Stubborn Links: interface

- **Module:**
  - Name: StubbornPointToPointLink
    instance sl
- **Events:**
  - **Request**: ⟨sl, Send | dest, m⟩
    - Request the transmission of message m to process dest
  - **Indication**:⟨sl, Deliver src, m⟩
    - deliver message m sent by process src
- **Properties:**
  - ***SL1, SL2***

Stubborn Links

- Properties
  - ***SL1. Stubborn delivery***: if a correct process p_i sends a message m to a correct process p_j, then p_j delivers m an infinite number of times
  - ***SL2. No creation:*** if a message m is delivered by some process p_j, then m was previously sent by some process p_i

Implementing Stubborn Links

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

Algorithm (sl)

```
Implements: StubbornLinks instance sl
Uses: FairLossLinks, instance all
• upon event ⟨sl, Init⟩ do
    • sent := ∅
    • startTimer(TimeDelay)

• upon event ⟨Timeout⟩ do
    • forall (dest, m) ∈ sent do
        • trigger ⟨fl, Send | dest, m⟩
    • startTimer(TimeDelay)

upon event ⟨sl, Send | dest, m⟩ do
    • trigger ⟨fll, Send | src, m⟩
    • sent := sent ∪ { (dest, m) }
upon event ⟨fll, Deliver | src, m⟩ do
    • trigger ⟨sl Deliver | src, m⟩
```

Implementing Stubborn Links

- Implementation
  - Use the Lossy link
  - Sender stores every message it sends in **sent**
  - It periodically resends all messages in **sent**
- Correctness
  - ***SL1. Stubborn delivery***
    - If process 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

Perfect Links

Channel failure modes

- Perfect Links
  - Channels that delivers any message sent exactly once

Perfect links: interface

- **Module:**
  - Name: PerfectPointToPointLink, **instance** pl
- **Events:**
  - **Request**: ⟨pl, Send | dest, m⟩
    - Request the transmission of message m to node dest
  - **Indication**: ⟨pl, Deliver | src, m⟩
    - deliver message m sent by node src
- **Properties:**
  - ***PL1, PL2, PL3***

Perfect links (Reliable links)

- ***Properties***
  - ***PL1. Reliable Delivery***: If p_i and p_j are correct, 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

Perfect links (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)

Perfect Link Implementation

- Implementation
  - Use Stubborn links
  - Receiver keeps log of all received messages in **Delivered**
    - Only deliver (perfect link Deliver) 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])

FIFO Perfect links (Reliable links)

- ***Properties***
  - ***PL1. Reliable Delivery***:
  - ***PL2. No duplication:***
  - ***PL3. No creation:*** No message is delivered unless it was sent
  - ***FFPL. Ordered Delivery***: if m_1 is sent before m_2 by p_i to p_j and m_2 is delivered by p_j then m_1 is delivered by p_j before m_2

Internet TCP vs. FIFO Perfect Links

- TCP provides reliable delivery of packets
- TCP reliability is so called “session based”
- Uses sequence numbers
  - ACK: “I have received everything up to byte X”
- Implementing Perfect Link abstraction on TCP requires reconciling messages between the sender and receiver when reestablishing connection after a session break

Default Assumptions in Course

- We assume perfect links (aka reliable) most of time in the course (unless specified otherwise)
- Roughly, reliable links ensure messages exchanged between correct 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

- Many algorithm for crash-recovery process model assume either a Stubborn link, or Logged perfect link

Timing Assumptions

Specification of a Service

- How to specify a distributed service (abstract)?
  - Interface (aka Contract, API)
    - Requests
    - Responses
  - Correctness Properties
    - Safety
    - Liveness
  - Model
    - Assumptions on failures
    - Assumptions on timing (amount of synchrony)

declarative
specification
“what”
aka problem

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

imperative,
many possible
“how”

Timing Assumptions

- Timing assumptions
  - Processes
    - bounds on time to make a computation step
  - Network
    - Bounds on time to transmit a message between a sender and a receiver
  - Clocks:
    - Lower and upper bounds on clock rate-drift and clock skew w.r.t. real time

Asynchronous Model and Causality

Asynchronous Systems

- No timing assumption on processes and channels
  - Processing time varies arbitrarily
  - No bound on transmission time
  - Clocks of different processes are not synchronized
- Reasoning in this model is based on which events may cause other events
  - Causality
- Total order of event not observable locally, no access to global clocks

Causal Order (happen before)

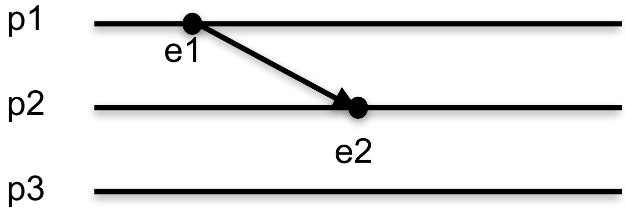
- The relation $\rightarrow_\beta$ on the events of an execution (or trace β), called also causal order, is defined as follows
  - If a occurs before b on the same process, then $a \rightarrow_\beta b$
  - If a is a send(m) and b deliver(m), then $a \rightarrow_\beta b$
  - $a \rightarrow_\beta b$ is transitive
    - i.e. If $a \rightarrow_\beta b$ and $b \rightarrow_\beta c$ then $a \rightarrow_\beta c$

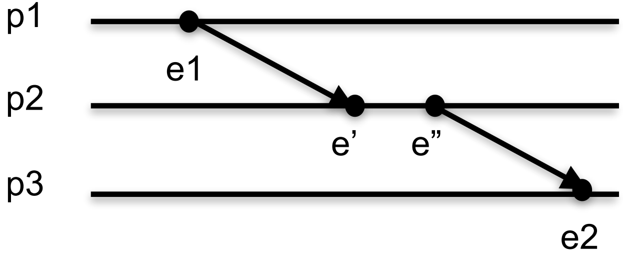
- Two events, a and b, are concurrent if not $a \rightarrow_\beta b$ and not $b \rightarrow_\beta a$
- $a \| b$

Causal Order (happen before)

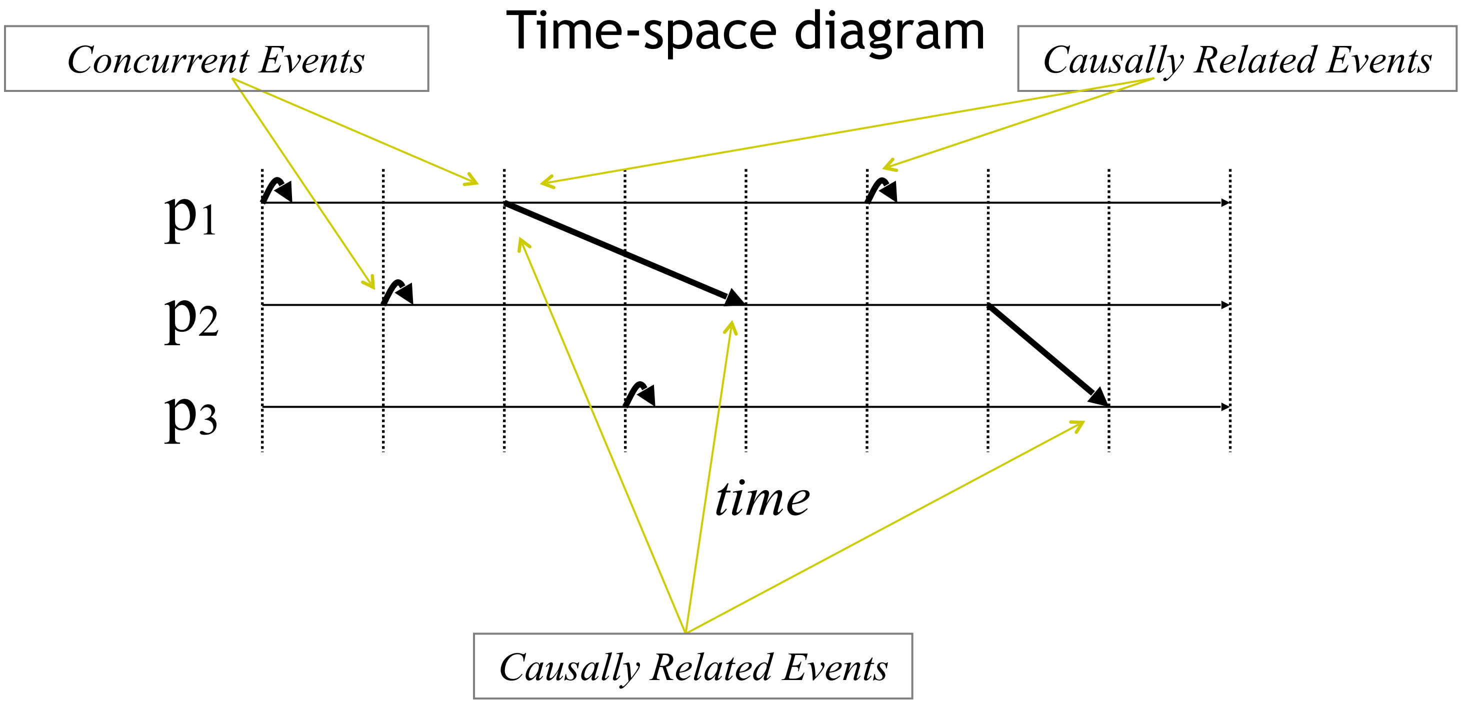
- The relation $\rightarrow_\beta$ on the events of an execution (or trace β), called also causal order, is defined as follows
    - If a occurs before b on the same process, then $a \rightarrow_\beta b$
    - If a is a send(m) and b deliver(m), then $a \rightarrow_\beta b$
    - $a \rightarrow_\beta b$ is transitive
        - i.e. If $a \rightarrow_\beta b$ and $b \rightarrow_\beta c$ then $a \rightarrow_\beta c$

- Two events, a and b, are concurrent if not $a \rightarrow_\beta b$ and not $b \rightarrow_\beta a$
- a||b





Example of Causally Related events



Similarity of executions

- The view of p_i in E, denoted $E|p_i$, is
  - the subsequence of execution E restricted to events and state of p_i
- Two executions E and F are similar w.r.t p_i if
  - $E|p_i = F|p_i$
- Two executions E and F are similar if
  - E and F are similar w.r.t every process

Equivalence of Executions

- **Computation Theorem:**
  - Let E be an execution (c_0,e_1,c_1,e_2,c_2,…), and V the trace of events (e_1,e_2,e_3,…)
  - Let P be a permutation of V, preserving causal order
    - P=(f_1, f_2, f_3…) preserves the causal order of V when for every pair of events $f_i \rightarrow_V f_j$ implies f_i is before f_j in P
  - Then E is similar to the execution starting in c_0 with trace P

Equivalence of executions

- If two executions F and E have the same collection of events, and their causal order is preserved, F and E are said to be similar executions, written $F{\sim}E$
  - F and E could have different permutation of events as long as causality is preserved!

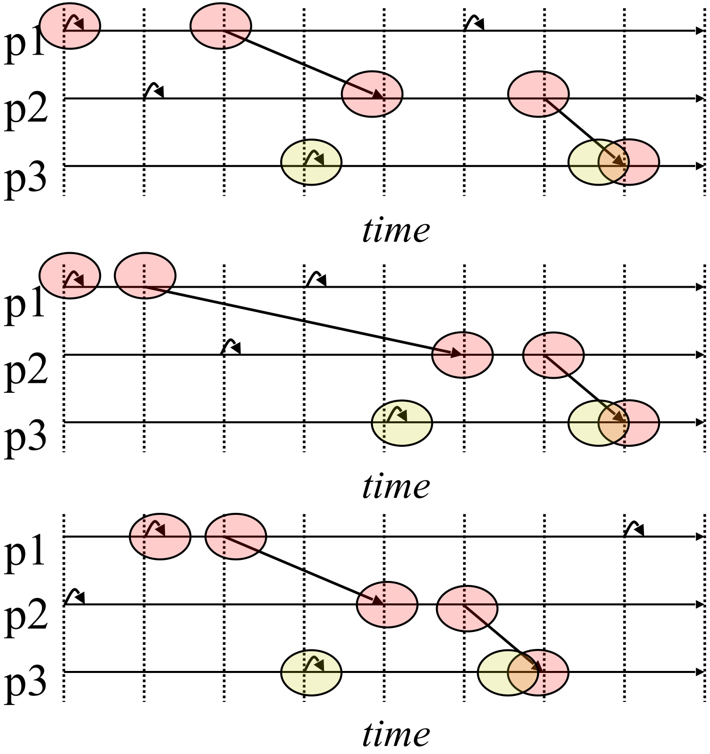
Computations

- Similar executions form equivalence classes where every execution in a class is similar to the other executions in the same class

- I.e. the following always holds for executions:
  - ~ is reflexive
    - I.e. a~ a for any execution
  - ~ is symmetric
    - I.e. If a~b then b~a for any executions a and b
  - ~ is transitive
    - If a~b and b~c, then a~c, for any executions a, b, c

- Equivalence classes are called computations of executions

Example of similar executions



🟨 🟥 *Same color ~ Causally related*

- All three executions are part of the same computation, as causality is preserved

Two important results (1)

- **Computation theorem gives two important results**
- **Result 1:** There is no algorithm in the asynchronous system model that can observe the order of the sequence of events (that can “see” the time-space diagram, or the trace) for all executions

Two important results (1)

- Proof:
  - Assume such an algorithm exists. Assume p knows the order in the final (repeated) configuration
  - Take two distinct similar executions of algorithm preserving causality
  - Computation theorem says their final repeated configurations are the same, then the algorithm cannot have observed the actual order of events as they differ

Two important results (2)

- Result 2: The computation theorem does not hold if the model is extended such that each process can read a local hardware clock

- Proof:
    - Similarly, assume a distributed algorithm in which each process reads the local clock each time a local event occurs
    - The final (repeated) configuration of different causality preserving executions will have different clock values, which would contradict the computation theorem

Synchronous Systems

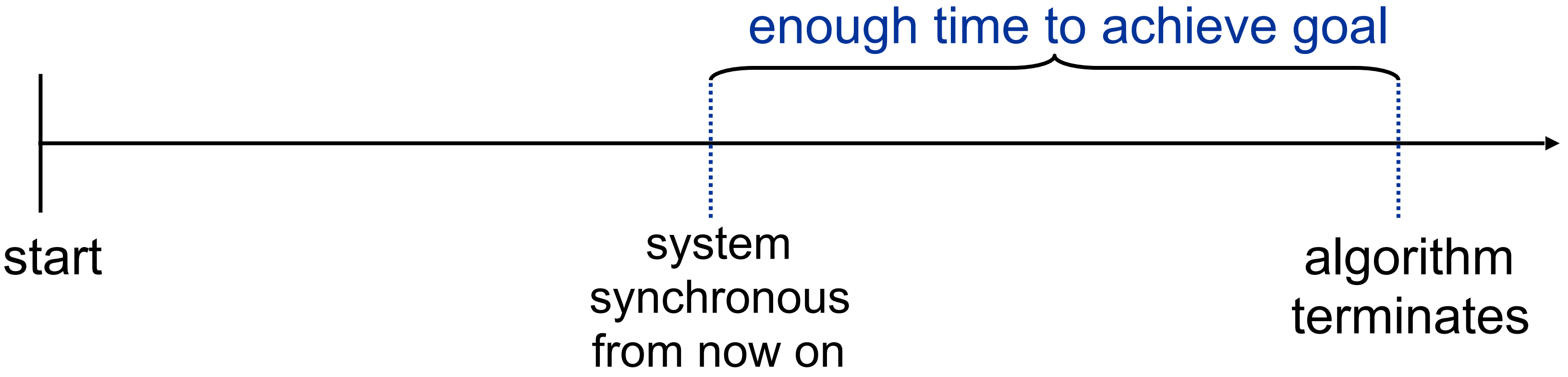
- Model assumes
  - Synchronous computation
    - Known upper bound on how long it takes to perform computation
  - Synchronous communication
    - Known upper bound on message transmission delay
  - Synchronous physical clocks
    - Nodes have local physical clock
    - Known upper bound clock-drift rate and clock skew
- Why study synchronous systems? [d]

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]

Partial Synchrony

- *Your algorithm will have a long enough time window, where everything behaves nicely (synchrony), so that it can achieve its goal*
  - Useful for proving liveness properties of algorithms



enough time to achieve goal

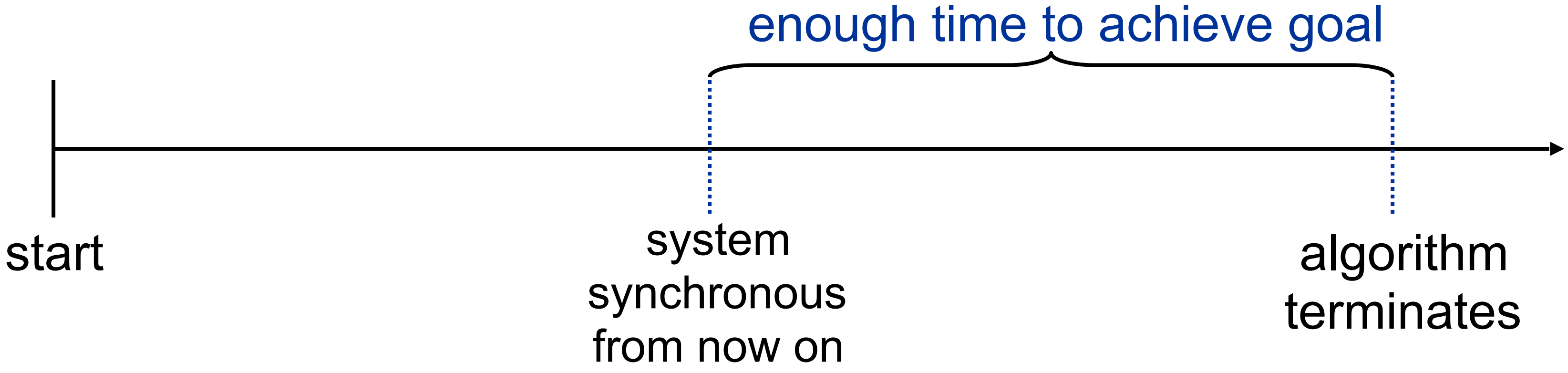
start

system synchronous from now on

algorithm terminates

Partial Synchrony

- Notice the time at which a system behaves synchronously is unknown
- To prove safety properties we need to assume that the system is asynchronous
- To prove liveness we use the partial synchrony assumption



enough time to achieve goal

start

system synchronous from now on

algorithm terminates

Timed Asynchronous Systems

- No timing assumption on processes and channels
  - Processing time varies arbitrarily
  - No bound on transmission time
- Bounds on Clocks drift-rate and clock skews
  - Interval clocks
  - At real-time t, clock of process P is in interval $(t-\rho, t+\rho)$
  - ρ depends on P

Logical Clocks

Logical Clocks

- A clock is function **t** from the events to a totally order set such that for events *a* and *b*
  - if $a \rightarrow b$ then $\mathbf{t}(a) < \mathbf{t}(b)$

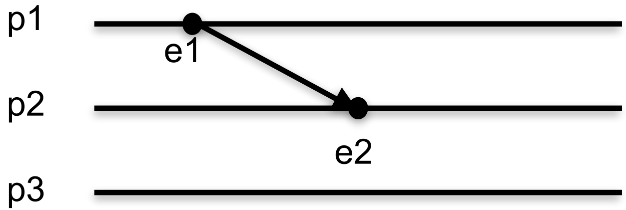
- We are interested in → being the happen-before relation

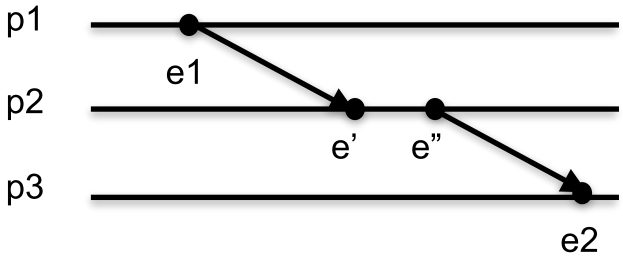
Causal Order (happen before)

- The relation $\rightarrow_\beta$ on the events of an execution (or trace β), called also causal order, is defined as follows
  - If a occurs before b on the same process, then $a \rightarrow_\beta b$
  - If a is a send(m) and b deliver(m), then $a \rightarrow_\beta b$
  - $a \rightarrow_\beta b$ is transitive
    - i.e. If $a \rightarrow_\beta b$ and $b \rightarrow_\beta c$ then $a \rightarrow_\beta c$

- Two events, a and b, are concurrent if not $a \rightarrow_\beta b$ and not $b \rightarrow_\beta a$
- $a || b$

Causal Order (happen before)





Observing Causality

- So causality is all that matters…
- …how to locally tell if two events are causally related?

Lamport Clocks at process p

- Each process has a local logical clock, kept in variable $\mathbf{t_p}$, initially $\mathbf{t_p} = 0$
  - A process p piggybacks ($\mathbf{t_p}$, p) on every message sent
- On internal event *a*:
  - $\mathbf{t_p} := \mathbf{t_p} + 1$; perform internal event *a*
- On send event message m:
  - $\mathbf{t_p} := \mathbf{t_p} + 1$; send(m, ($\mathbf{t_p}$, p))
- On delivery event *a* of m with timestamp (t_q, q) from q:
  - $\mathbf{t_p} := \max(\mathbf{t_p}, \mathbf{t_q}) + 1$; perform delivery event *a*

Lamport Clocks (2)

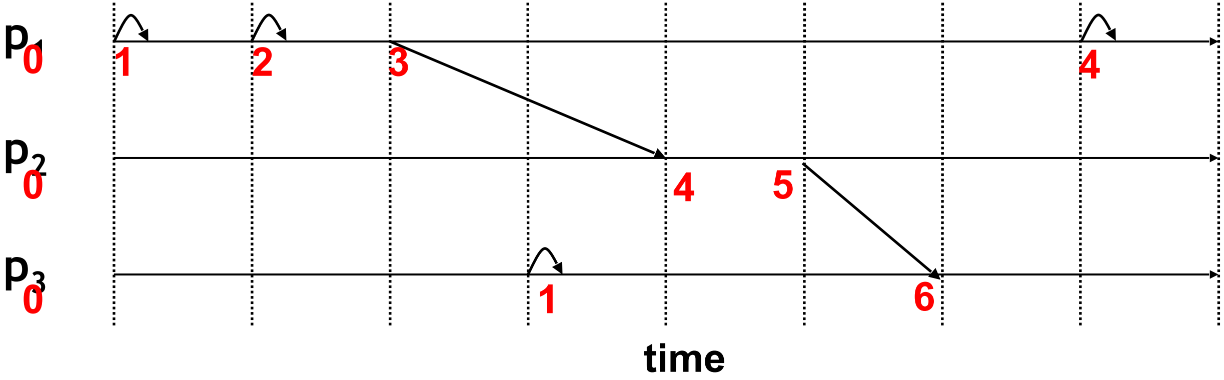
- Observe the timestamp (t, p) is unique
- Comparing two timestamps (t_p,p) and (t_q,q)
  - $(t_p,p)<(t_q,q)$ iff $(t_p<t_q$ or $(t_p=t_q$ and $p<q))$
  - i.e. break ties using process identifiers
  - e.g. $(5,p_5) < (7,p_2)$, $(4,p_2) < (4,p_3)$

Lamport Clocks (2)

- Lamport logical clocks guarantee that:
  - If $a \rightarrow_\beta b$, then $\mathbf{t}(a) < \mathbf{t}(b)$,
    - where $\mathbf{t}(a)$ is Lamport clock of event a

- events a and b are on the same process p, t_p is strictly increasing, so if a is before b, then t(a) < t(b)
- a is a send event with t_q and b is deliver event, t(b) is at least one larger than t_q (t(a))
- transitivity of t(a) < t(b) < t(c) implies the transitivity condition of the happen before relation

Lamport logical clocks



- Lamport logical clocks guarantee that:
  - If $a \rightarrow_\beta b$, then $\mathbf{t}(a) < \mathbf{t}(b)$,
  - if $\mathbf{t}(a) \geq \mathbf{t}(b)$, then not $(a \rightarrow_\beta b)$

Vector Clocks

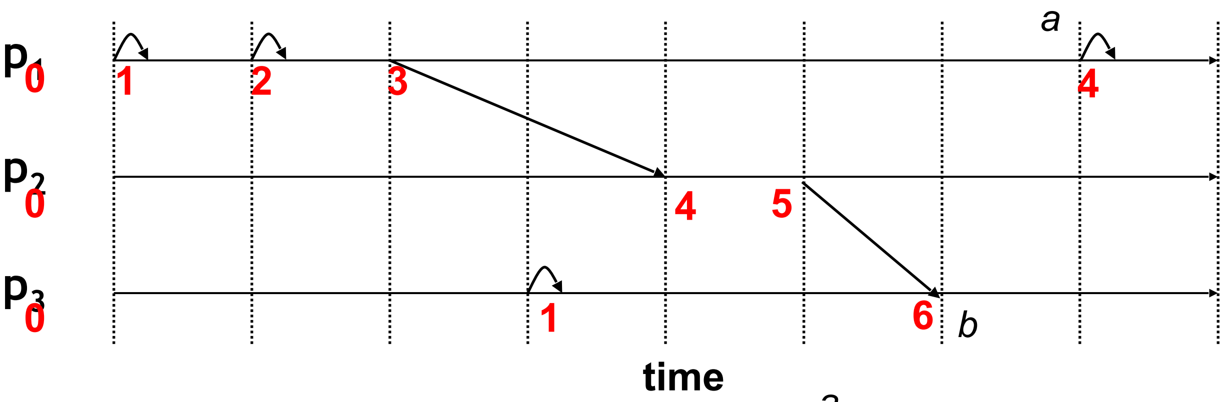
Vector clocks

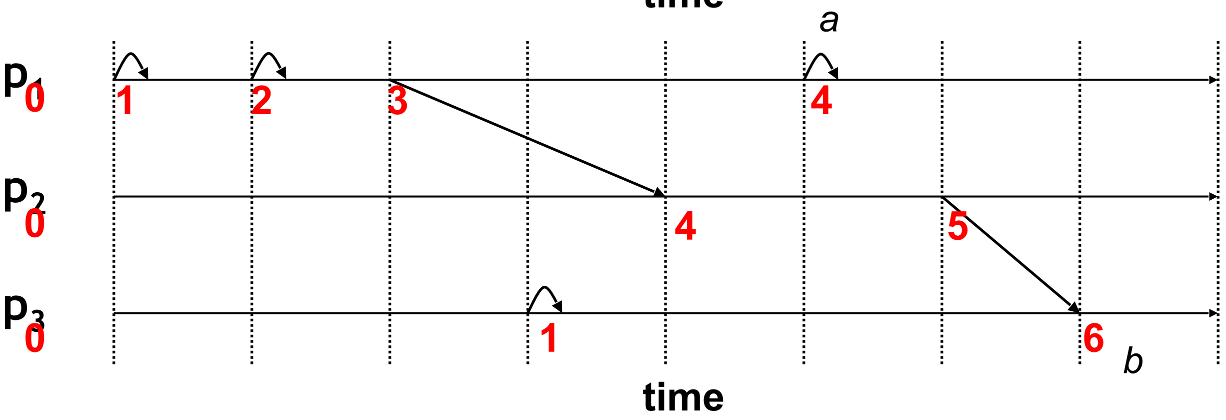
- The happen-before relation is a partial order
- In contrast logical clocks are total
  - Information about non-causality is lost
    - We cannot tell by looking to the timestamps of event *a* and *b* whether there is a causal relation between the events, or they are concurrent
- Vector clocks guarantee that:
  - if $\mathbf{v}(a) < \mathbf{v}(b)$ then $a \rightarrow_{\beta} b$, in addition to
  - if $a \rightarrow_{\beta} b$ then $\mathbf{v}(a) < \mathbf{v}(b)$
    - where $\mathbf{v}(a)$ is a vector clock of event *a*

Non-causality and Concurrent events

- Two events *a* and *b* are concurrent ($a \parallel_\beta b$) in an execution E (trace(E) = β) if
  - **not** $a \rightarrow_\beta b$ and **not** $b \rightarrow_\beta a$
- Computation theorem implies that if ($a \parallel_\beta b$) in β then there are two executions (with traces β_1 and β_2) that are similar where *a* occurs before *b* in β_1, *b* occurs before *a* in β_2

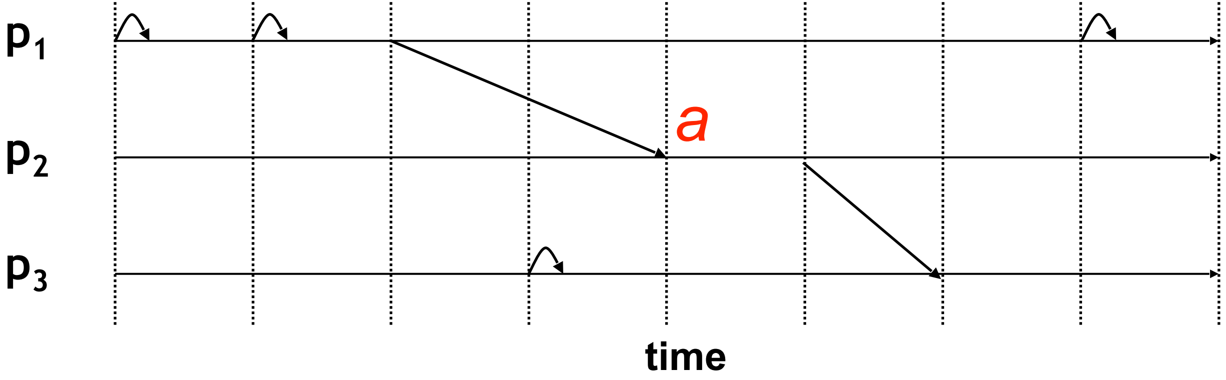
Non-causality and Concurrent events





Vector clock definition

- Vector clock for an event ***a***
  - **v**(a) = ($x_1, \ldots, x_n$)
  - x_i is the number of events at p_i that happens-before a
  - for each such event e: e → a



Vector Timestamps

- Processes $p_1, \ldots, p_n$
- Each process p_i has local vector **v** of size **n** (number of processes)
  - **v**[i] = 0 for all i in 1…n
  - Piggyback **v** on every sent message
- For each transition (on each event) update local **v** at p_i:
  - **v**[i] := **v**[i] + 1 (internal, send or deliver)
  - **v**[j] := max(**v**[j], $\mathbf{v_q}$[j]), for all j ≠ i (deliver)
    - where $\mathbf{v_q}$ is clock in message received from process q

Comparing Vector Clocks

- $v_p \leq v_q$ iff
  - $v_p[i] \leq v_q[i]$ for all i
- $v_p < v_q$ iff
  - $v_p \leq v_q$ and for some i, $v_p[i] < v_q[i]$
- v_p and v_q are concurrent ($v_p \| v_q$) iff
  - not $v_p < v_q$, and not $v_q < v_p$

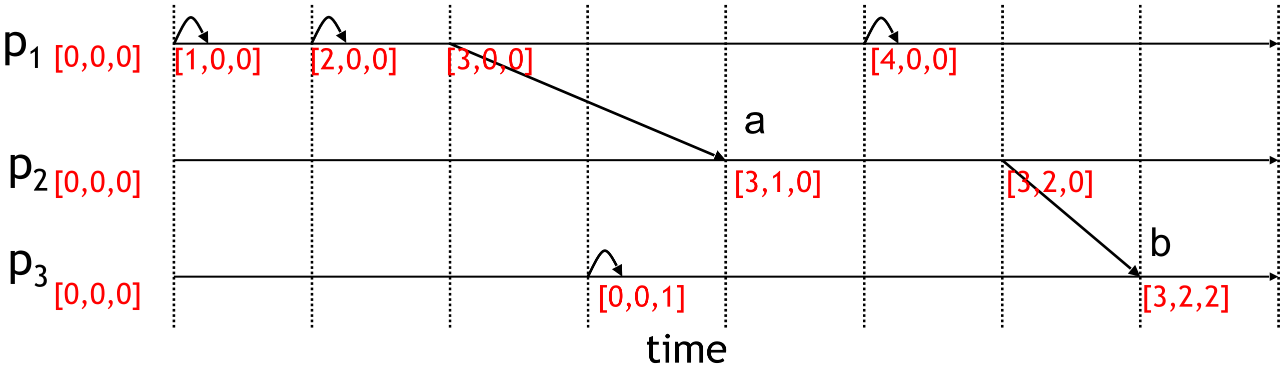
$(3,0,0) \leq (3,1,0)$

$[3,0,0] < [3,1,0]$

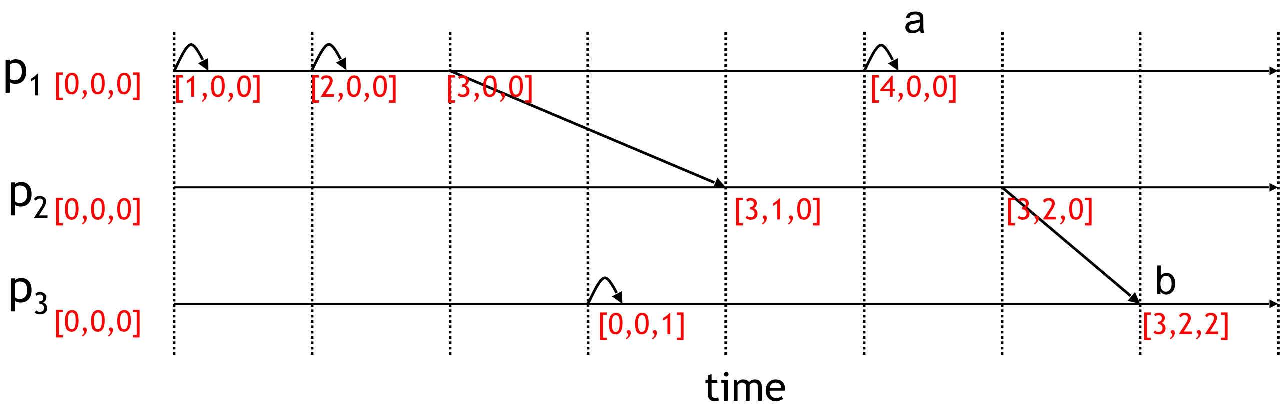
$[3,1,0] <> [4,0,0]$

- Vector clocks guarantee
  - If $v(a) < v(b)$ then $a \rightarrow b$, and
  - If $a \rightarrow b$, then $v(a) < v(b)$
    - where v(a) is the vector clock of event a

Example of Vector Timestamps

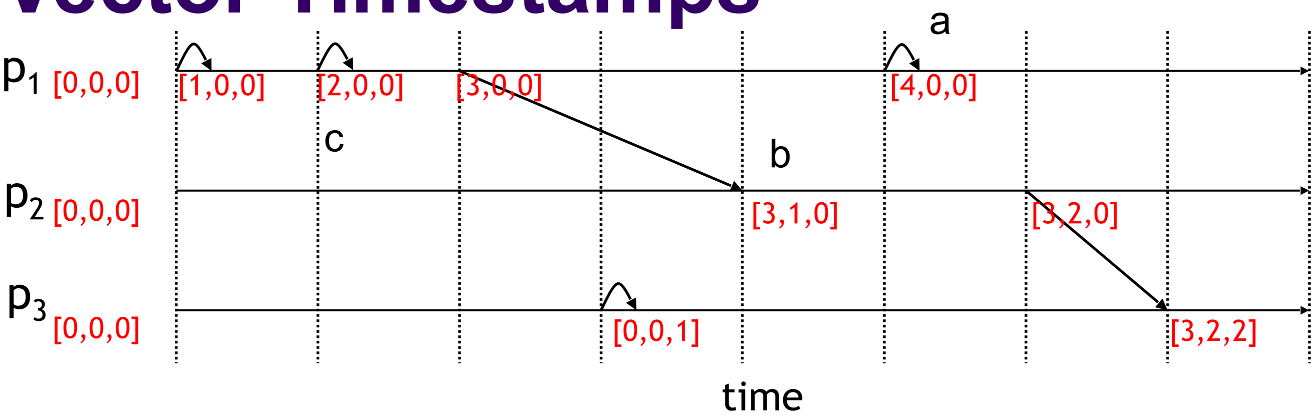


v(a) < v(b) implies a → b



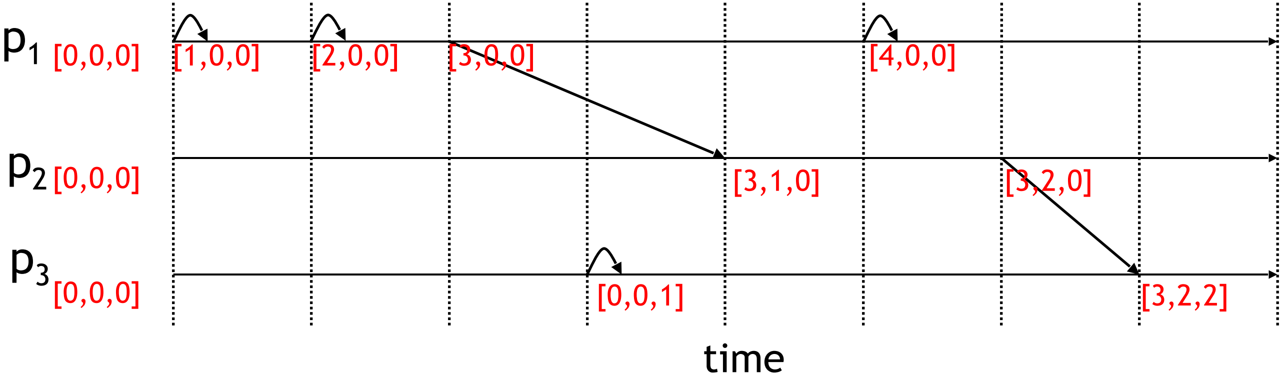
v(a) <> v(b) implies a || b

Vector Timestamps



- For any events a and b, and trace β :
  - **v**(a) and **v**(b) are incomparable if and only if a||b
  - **v**(a) < **v**(b) if and only if a → b

Example of Vector Timestamps



Great! But cannot be done with smaller vectors than size n, for n nodes

Partial and Total Orders

- Only a partial order or a total order? [d]
  - the relation $\rightarrow_\beta$ on events in executions
    - Partial: $\rightarrow_\beta$ doesn't order concurrent events
  - the relation < on Lamport logical clocks
    - Total: any two distinct clock values are ordered (adding pid)
  - the relation < on vector timestamps
    - Partial: timestamp of concurrent events not ordered

Logical clock vs. Vector clock

- **Logical clock**
  - If $a \rightarrow_{\beta} b$ then $t(a) < t(b)$ (1)

- **Vector clock**
  - If $a \rightarrow_{\beta} b$ then $v(a) < v(b)$ (1)
  - If $v(a) < v(b)$ then $a \rightarrow_{\beta} b$ (2)

- Which of (1) and (2) is more useful? [d]

- What extra information do vector clocks give? [d]