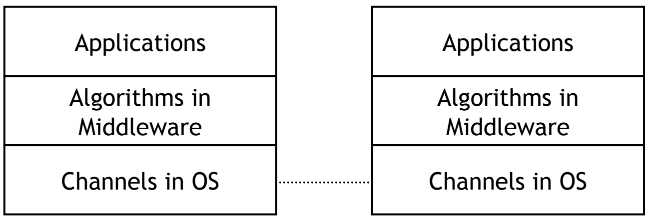
Specification and Implementation of Distributed Systems

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

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 (eg TCP, UDP)



Need of 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

2/20/12 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

2/20/12 4

Event-based Component Model

The Model

- Each node models a sequential program
- Assume a global clock
- At each tick either a node takes a step
- A node: Computation Step
  a) Performs some computation (local)
  b) Either sends or receives one message from some one node (global)
- Or, a 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]

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):

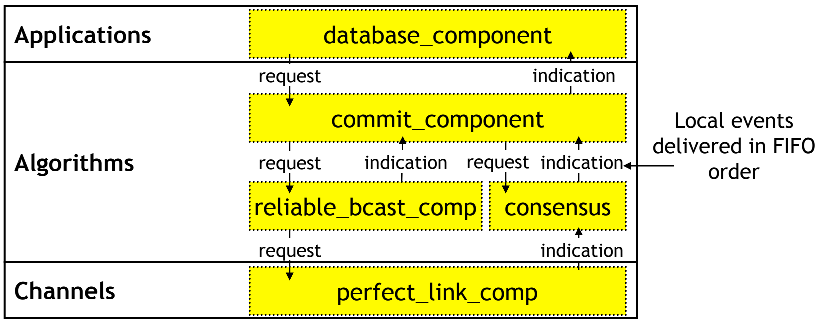
```
upon event <RequestEvent, attr1, attr2,...> do
        // local computation
        trigger <ResponseEvent, attr3, attr4,...>
```


Event-based Programming

- Events can be almost anything
  - Messages (most of the time)
  - Timers
  - Conditions (e.g. x==5 & y<9)
- Three main types of events
  - Requests (flow downward)
  - Indications (like responses flow upward)
  - Confirmations (Special type of indication, like an OK or ACK)

Modules on a Node

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



Send and Receive Instructions

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

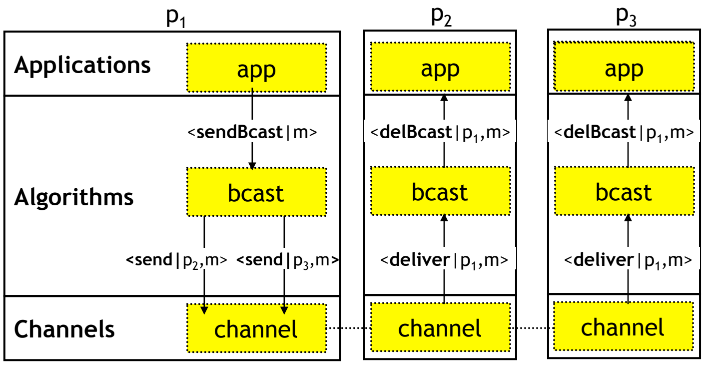
    trigger <*sendBcast* | *dest*, [data1, data2, …] >

- 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**

Example

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



Specification

Specification of a Service

- How to specify a distributed service?
  - Interface (aka Contract, API)
    - Requests
    - Responses
  - Correctness Properties
    - Safety
    - Liveness
  - Model
    - Assumptions on failures
    - Assumptions on timing (amount of synchrony)
  - Implementation
    - Composed of other services
    - Adheres to interface and satisfies correctness
    - Has internal events

declarative specification "what" aka problem

imperative, many possible "how"

Simple Interface Example (1)

Module 1.1 Interface of a printing module

Module:

Name: Print.

Events:

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

Confirmation:$\langle$ *PrintConfirm* | rqid $\rangle$: Used to confirm that the printing request with identifier rqid succeeded.

Simple Interface Example (2)

Algorithm 1.1 Printing service

Implements:
Print.

upon event ⟨ *PrintRequest* | rqid, str ⟩ **do**
print str;
trigger ⟨ *PrintConfirm* | rqid ⟩;

Simple Interface Example (3)

Module 1.2 Interface of a bounded printing module

Module:

Name: BoundedPrint.

Events:

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

Confirmation:⟨ *PrintStatus* | rqid, status ⟩: Used to return the outcome of the printing request: Ok or Nok.

Indication:⟨ *PrintAlarm* ⟩: Used to indicate that the threshold was reached.

Simple Interface Example (4)

Algorithm 1.2 Bounded printer based on (unbounded) printing service

Implements:
BoundedPrint.

Uses:
Print.

upon event ⟨ *Init* ⟩ **do**
bound := PredefinedThreshold;

<***Init***> automatically generated upon component creation

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

upon event ⟨ *PrintConfirm* | rqid ⟩ **do**
trigger ⟨ *PrintStatus* | rqid, Ok ⟩;

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

Correctness Example

- Correctness of You in SINF2345
  - 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
    - More than one direction should never have a green light at same time
  - Liveness
    - Every direction should eventually get a green light

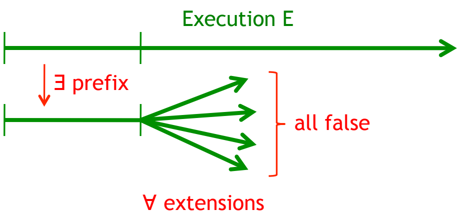
Safety & Liveness All That Matters

- 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

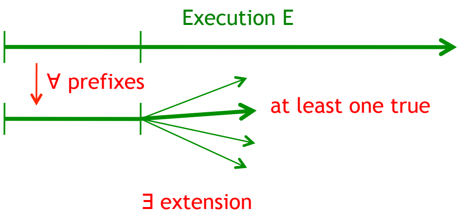
Prefixes and Extensions

- The prefix of execution E is the first *k* (for some *k*>0) configurations and events of E
  - 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

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

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,
  - For every prefix F of E, there exists an extension of 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
  - 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", "must", "at least"
  - Eventually means at some (often unknown) point in "future"

- Liveness is often just "termination"

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]

Node Behavior (failures)

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

Node Failures

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

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 nodes are not allowed to recover? [d]

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

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)

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

Fault-tolerance Hierarchy (2)

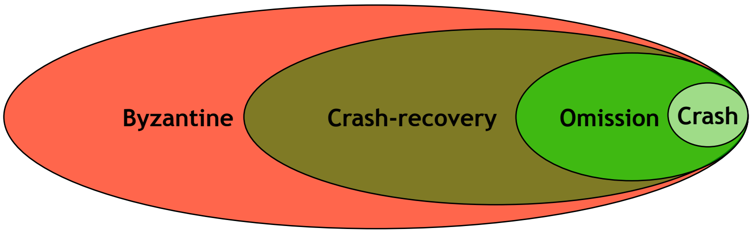
- Assume a special case of crash-recovery
  - Where nodes use only stable storage (no other memory)
- In this case crash-recovery is identical to omission
  - Crashing, recovering, and reading last state from storage
  - Just same as omitting send/recv while being crashed

- 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

Fault-tolerance Hierarchy (3)

- Crash-recovery is a special case of Byzantine
  - Since Byzantine allows anything
- Byzantine tolerance → crash-recovery tol.
  - Crash-recovery → omission, omission → crash

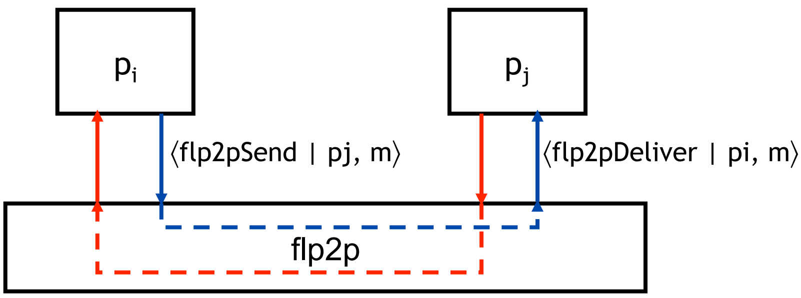


Channel Behavior (failures)

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

Fair Loss Links



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*.**

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

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***

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

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
  $E = <c_0, e_1, c_1, e_2, c_2, \ldots>$
- *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** ⟨Init⟩ **do**
  - sent := ∅
  - startTimer(TimeDelay)

- **upon event** ⟨Timeout⟩ **do**
  - **forall** (dest, m) ∈ sent **do**
    - **trigger** ⟨flp2pSend | dest, m⟩
  - startTimer(TimeDelay)

Algorithm (sl) (2)

- **upon event** ⟨sp2pSend | dest, m⟩ **do**
  - sent := sent ∪ { (dest, m) }

- **upon event** ⟨flp2pDeliver | src, m⟩ **do**
  - **trigger** ⟨sp2pDeliver src, m⟩

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***

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

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)

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])

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⟩

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

TCP vs perfect links

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

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

Timing Assumptions

Timing Assumptions

- Timing 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

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]

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]

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 (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