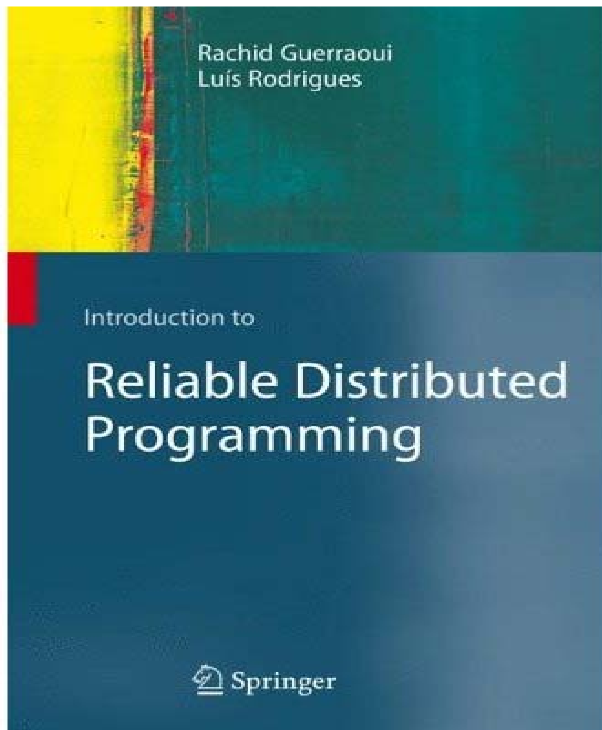
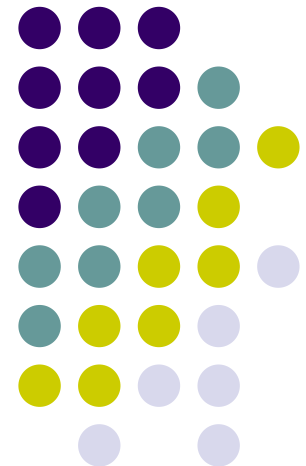
Distributed Algorithms for Building Reliable Systems

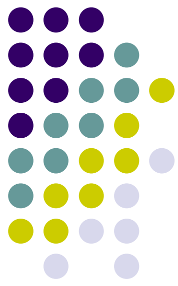


Seif Haridi

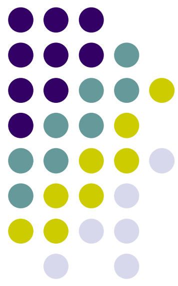
Overview



Algorithms (history)



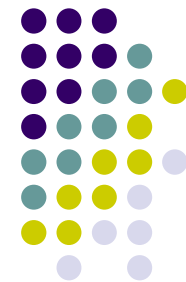
- M. Al-Khawarizmi ~9th century in Baghdad: inventor of the zero, the decimal system, Arithmetic and Algebra
- Calculated (with M. Al. Rachid) the circumference and volume of planets (including the earth): the first significant program ☺



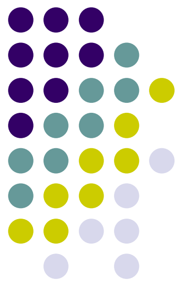
In short

- We study algorithms for ***distributed*** systems: a new way of thinking about algorithms
- Whereas a centralized algorithm is the soul of a computer, a distributed algorithm is the soul of a ***society*** of computers

Distributed algorithms (history)



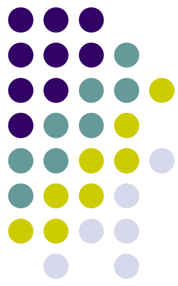
- E. Dijkstra (concurrent os)~60's
- L. Lamport: "a distributed system is one that stops your application because of a machine you have never heard from" ~70's
- J. Gray (transactions) ~70's
- N. Lynch (consensus) ~80's
- Birman, Schneider, Toueg – Cornell – (this course) ~90's



Important

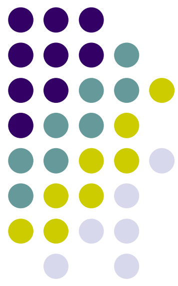
- We study here only algorithms based on ***message passing***
- Our context is a set of computers (processes) connected by a network

Overview



- **(1) *Why?*** Motivation
- (2) ***Where?*** Between the network and the application
- (3) ***How?*** (3.1) Specifications, (3.2) assumptions, and (3.3) algorithms

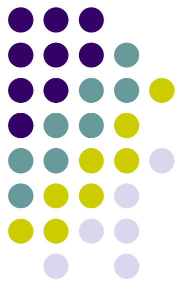
A distributed system



B

A

C

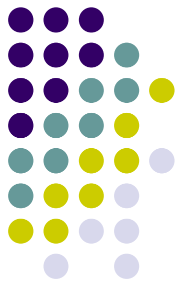


Clients-server

Client A

Client B

Server



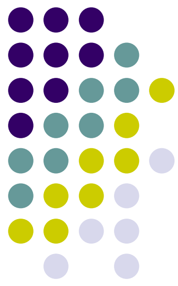
Multiple servers (genuine distribution)

Server B

Server A

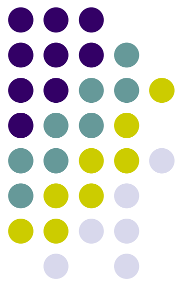
Server C

Applications



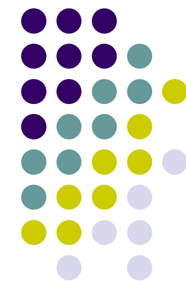
- Military and traffic control
- Finances: transactions, e-banking, stock-exchange
- Reservation systems
- In general mission critical systems

The optimistic view



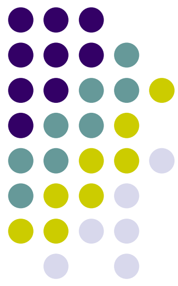
- By having multiple machines we get
  - Concurrency $\Rightarrow$ speed (load-balancing)
  - Partial failures $\Rightarrow$ high-availability

The pessimistic view



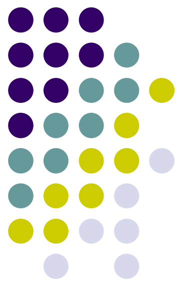
- By having multiple machines we get
  - Concurrency (interleaving) $\Rightarrow$ incorrectness
  - Partial failures $\Rightarrow$ incorrectness

Overview



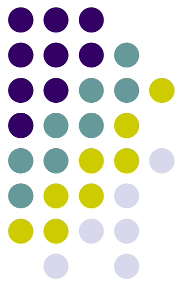
- (1) ***Why?*** Motivation
- **(2)** ***Where?*** Between the network and the application
- (3) ***How?*** (3.1) Specifications, (3.2) assumptions, and (3.3) algorithms

Distributed systems



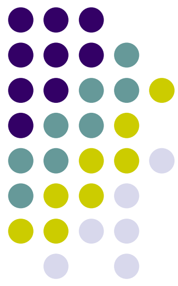
Applications	Applications
Algorithms	Algorithms
Channels	Channels

Distributed systems



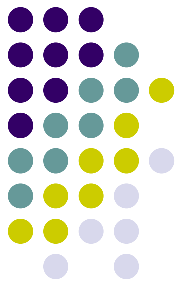
- The application needs underlying services for many-to-many interaction
- Network protocols are not enough
  - Reliability guarantees (e.g., TCP) are only offered for communication among pairs of processes, i.e., *one-to-one* communication (*client-server*)

Content of this course



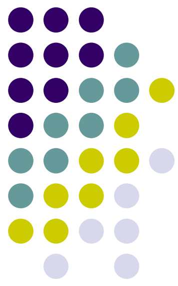
Reliable broadcast
Causal order broadcast
Shared memory
Consensus
Total order broadcast
Atomic commit
Terminating reliable broadcast
Group membership

Reliable distributed services

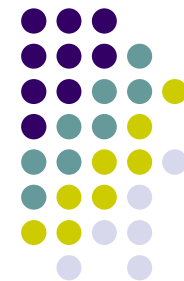


- Example 1: ***reliable broadcast***
  - Ensure that a message sent to a group of processes is received (delivered) by all or none
- Example 2: ***atomic commit***
  - Ensure that the processes reach decision on whether to commit or abort a transaction

Underlying elements of the model



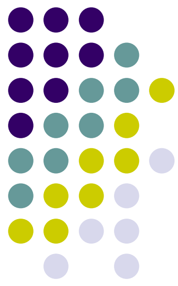
- (1): ***processes*** (abstracting computers)
- (2): ***channels*** (abstracting networks)
- (3): ***failure detectors*** (abstracting time)



Processes

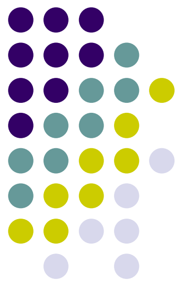
- The distributed system is made of a finite set of processes
  - each process models a sequential program
- Processes are denoted by $p_1,..p_N$ or p, q, r
- Processes have unique identities and know each other
- Every pair of processes is connected by a link through which the processes exchange messages
- Messages are uniquely identified (e.g. sender process + time-stamp)

Processes



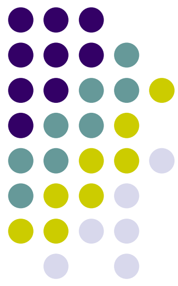
- A distributed algorithm is viewed as a collection of automata, one per process
- The automaton defines the way a process executes its computation steps
- An execution of a distributed algorithm is a linear sequence of steps each executed by one process
- Even if two steps (by different processes) could happen at the same time instance, we view them as if they happen at two different time instances

Processes



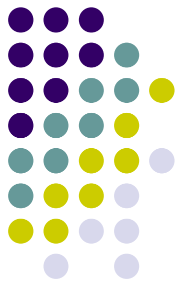
- A process step consists of
  - Receiving (delivering) a message from another process (global event)
  - Executing a local computation (local event)
  - Sending a message to some process (global event)
- NB. Any of these substeps could be empty

Processes



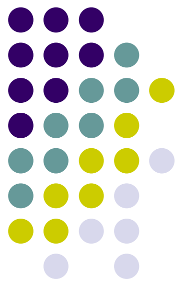
- The program of a process is made of a finite set of modules/components
  - deployed at runtime as component instances
  - typically organized at runtime as a software stack
- Component instances within the same process interact by exchanging events

Processes



- A module is described by
  - A set of local variables and an initialization procedure, called when the module is deployed as a component instance
  - A set of event-handler specifications
- Event-based specs
- **upon event** ⟨Event1 att1, att2, … ⟩ **do**
  - // something
  - **trigger** ⟨Event2 att1, att2, … ⟩

Components of a process



Applications

indication

request (deliver)

Algorithmic Components

indication

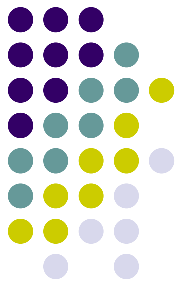
request (deliver)

indication

request (deliver)

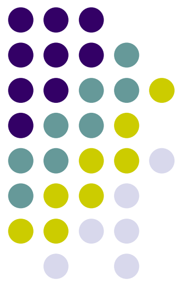
Channels

Overview



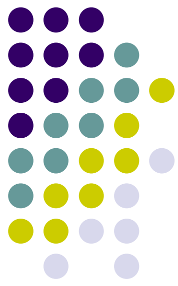
- (1) *Why?* Motivation
- (2) *Where?* Between the network and the application
- **(3) *How?*** (3.1) Specifications, (3.2) assumptions, and (3.3) algorithms

Approach



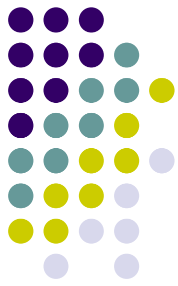
- ***Specifications***: What is the service? i.e., the problem ~ liveness + safety properties
- ***Assumptions***: What is the model, i.e., the power of the adversary?
- ***Algorithms***: How do we implement the service? Where are the bugs (proof)? What cost?

Overview



- (1) ***Why?*** Motivation
- (2) ***Where?*** Between the network and the application
- **(3) *How?* (3.1)** Specifications, (3.2) assumptions, and (3.3) algorithms

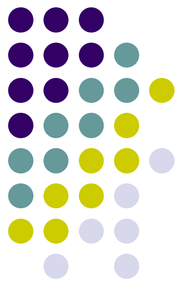
Liveness and safety



- ***Safety*** is a property which states that nothing bad should happen
  - A property that always holds after any partial execution
- ***Liveness*** is a property which states that something good should happen

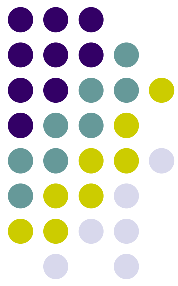
- Any specification can be expressed in terms of liveness and safety properties (Lamport and Schneider)

Liveness and safety



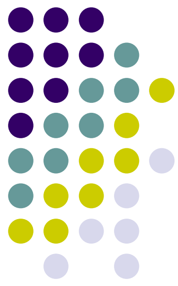
- Example: *Tell the truth*
  - Having to say something is *liveness*
  - Not lying is *safety*

Specifications



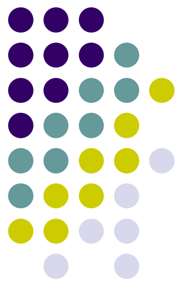
- Example 1: ***reliable broadcast***
  - Ensure that a message sent to a group of processes is received by all or none
- Example 2: ***atomic commit***
  - Ensure that the processes reach decision on whether to commit or abort a transaction

Overview



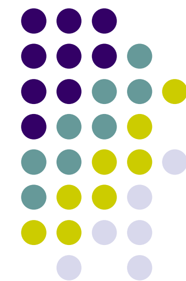
- (1) ***Why?*** Motivation
- (2) ***Where?*** Between the network and the application
- **(3) *How?*** (3.1) Specifications, **(3.2)** assumptions, and (3.3) algorithms

Assumptions



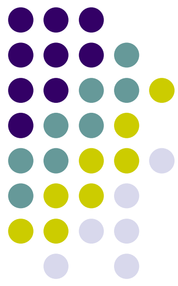
- **3.2.1** Assumptions on processes and channels (defines the processors and network failure model)
- 3.2.2 Failure detection (defines the timing model)

Process failure modes



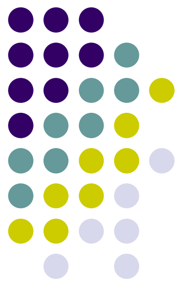
- A process either executes the algorithm assigned to it (steps) or fails
- Two broad categories of failures are mainly considered:
  - ***Omissions****:* the process omits to send messages it is supposed to send (distracted)
  - ***Arbitrary****:* the process sends messages it is not supposed to send (malicious or Byzantine)

Processes



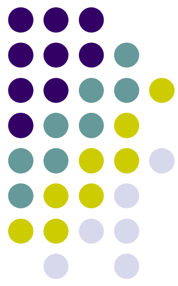
- ***Crash-recovery:*** general case of omissions
  - A process loses its volatile storage state when it crashes (amnesia)
  - The process has a stable storage that it does not lose
  - NB. A challenge here is to devise algorithms that use few accesses to stable storage

Processes

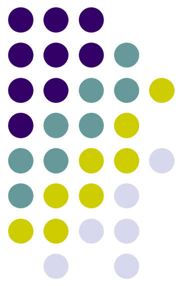


- ***Crash-stop:*** a more specific case of omissions
  - A process that omits a message to a process, omits all subsequent messages to all processes (permanent distraction): it crashes
  - No need for stable storage

Processes



- By default, we shall assume a ***crash-stop*** model throughout this course; that is, unless specified otherwise: processes fail only by crashing (no recovery)
- A ***correct*** process is a process that does not fail (that does not crash)



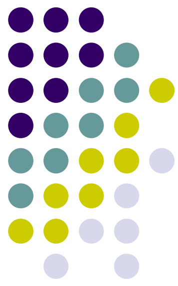
Failure modes

Crashes

Omissions

Crashes & Recoveries

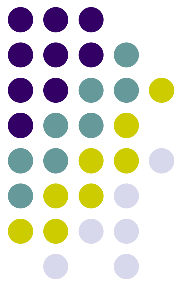
Arbitrary



Fair-loss links

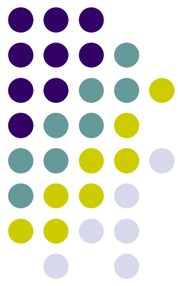
- ***FL1. Fair-loss***: If a message is sent infinitely often by p_i to p_j , and neither p_i or p_j crashes, then m is delivered infinitely often by p_j
- ***FL2. Finite duplication:*** If a message is sent a finite number of times by p_i to p_j, it is delivered a finite number of times by p_j
- ***FL3. No creation:*** No message is delivered unless it was sent

Fair-loss links: Interfaces



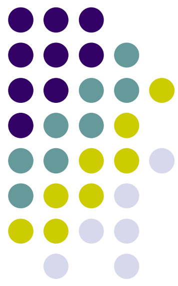
- **Module:**
  - Name: flp2p (FairLossPointToPoint)
- **Events:**
  - **Request (input event)**: ⟨flp2pSend dest, m⟩: request the transmission of message m to process dest
  - **Indication (output event)**:⟨flp2pDeliver src, m⟩: deliver message m sent by process src
- **Properties:**
  - ***FL1. FL2. FL3.***

Stubborn links



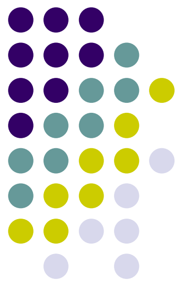
- ***SL1. Stubborn delivery***: if a process p_i sends a message m to a correct process p_j, and p_i does not crash, then p_j delivers m an infinite number of times
- ***SL2. No creation:*** No message is delivered unless it was sent

Stubborn links: interface



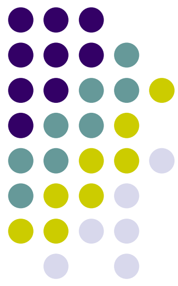
- **Module:**
  - Name: sp2p (StubbornPointToPoint)
- **Events:**
  - **Request**: ⟨sp2pSend dest, m⟩: request the transmission of message m to process dest
  - **Indication**:⟨sp2pDeliver src, m⟩: deliver message m sent by process src
- **Properties:**
  - ***SL1. SL2.***

Algorithm (sl)

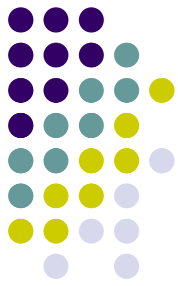


- **Implements:** StubbornLinks (sp2p)
- **Uses:** FairLossLinks (flp2p)
- **upon event** ⟨Init⟩ **do**
  - sent := ∅
  - startTimer(TimeDelay)
- **upon event** ⟨Timeout⟩ **do**
  - **forall** (dest, m) ∈ sent **do**
    - **trigger** ⟨flp2pSend dest, m⟩
  - startTime(TimeDelay)
- **upon event** ….

Algorithm (sl)

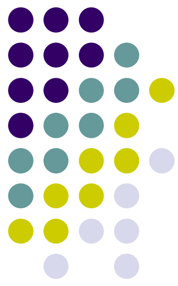


- **Implements:** StubbornLinks (sp2p).
- **Uses:** FairLossLinks (flp2p).
- ...
- ...
- **upon event** ⟨sp2pSend dest, m⟩ **do**
  - **trigger** ⟨flp2pSend dest, m⟩
  - sent := sent ∪ { (dest, m) }
- **upon event** ⟨flp2pDeliver src, m⟩ **do**
  - **trigger** ⟨sp2pDeliver src, m⟩



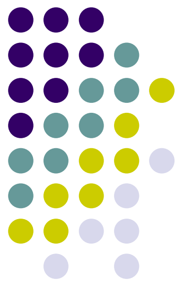
Reliable (Perfect) links

- ***Properties***
  - ***PL1. Validity***: If pi and pj are correct, then every message sent by pi to pj is eventually delivered by pj
  - ***PL2. No duplication:*** No message is delivered (to a process) more than once
  - ***PL3. No creation:*** No message is delivered unless it was sent



Reliable Links

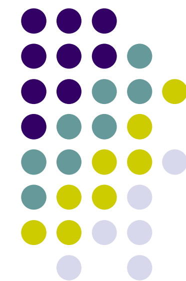
- ***Remarks.***
  - PL1 is a liveness property whereas PL2 and PL3 are safety properties
  - We talk here about the no duplication of the delivery and not of the message, i.e., PL3 does not imply PL2



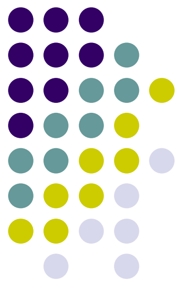
Algorithm (pl)

- **Implements:** PerfectLinks (pp2p).
- **Uses:** StubbornLinks (sp2p).
- **upon event** ⟨Init⟩ **do** delivered := $\emptyset$
- **upon event** ⟨pp2pSend dest, m⟩ **do**
  - **trigger** ⟨sp2pSend dest, m⟩
- **upon event** ⟨sp2pDeliver src, m⟩ **do**
  - **if** m $\notin$ delivered **then**
    - delivered := delivered $\cup$ { m }
    - **trigger** ⟨pp2pDeliver src, m⟩

Reliable links

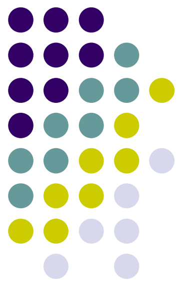


- We shall assume reliable links (also called perfect) throughout this course (unless specified otherwise)
- Roughly speaking, reliable links ensure that messages exchanged between correct processes are not lost
- NB. Messages are uniquely identified and the message identifier includes the sender's identifier
- Stubborn links are used mostly in the crash-recovery algorithms



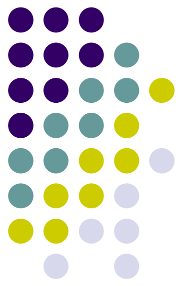
Overview

- (1) ***Why?*** Motivation
- (2) ***Where?*** Between the network and the application
- **(3) *How?*** (3.1) Specifications, **(3.2)** assumptions, and (3.3) algorithms
  - 3.2.1 Processes and links
  - **3.2.2** Failure Detection



Failure Detection

- A ***failure detector*** is a distributed oracle that provides processes with suspicions about crashed processes
- It is implemented using (i.e., it encapsulates) timing assumptions
- According to the timing assumptions, the suspicions can be accurate or not

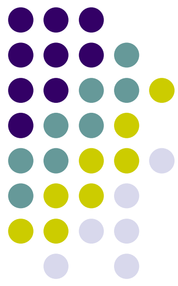


Failure Detection

Implementation:

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

Failure Detection



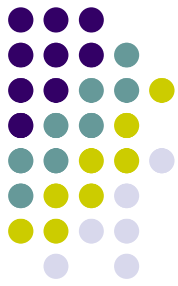
Perfect: **(P)**

- *Strong Completeness:* Eventually, every process that crashes is permanently suspected by every correct process
- *Strong Accuracy:* No process is suspected before it crashes

Eventually Perfect: **(◊P)**

- *Strong Completeness*
- *Eventual Strong Accuracy:* Eventually, no correct process is ever suspected

Timing assumptions



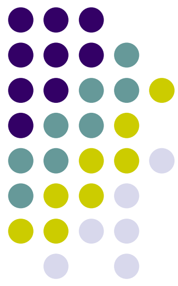
Synchronous:

- *Processing:* the time it takes for a process to execute a step is bounded and known
- *Delays:* there is a known upper bound limit on the time it takes for a message to be received
- *Clocks:* the drift between a local clock and the global real time clock is bounded and known

Asynchronous: no assumption

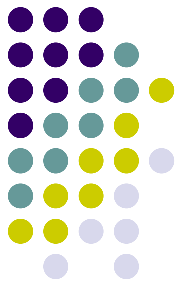
Eventually Synchronous: the timing assumptions hold eventually

Perfect Failure Detector (P)



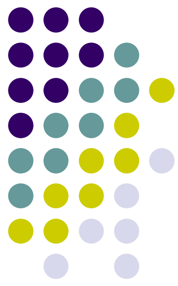
- A failure detector module is defined by events and properties
- ***Events***
  - Indication: ⟨crash p⟩
- ***Properties:***
  - Strong Completeness
  - Strong Accuracy

Algorithm: Exclude on Timeout



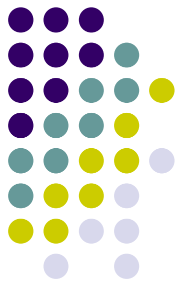
- **Implements:** PerfectFailureDetector (P)
- **Uses:** PerfectPointToPointLinks (pp2p)
- **upon event** ⟨Init⟩ **do**
  - alive := Π; detected := $\emptyset$; startTimer (TimeDelay)
- **upon event** ⟨ Timeout ⟩ **do**
  - …
  - …

Algorithm: Exclude on Timeout



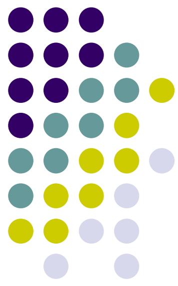
- **upon event** ⟨ Timeout ⟩ **do**
  - **forall** $p_i \in \Pi$ **do**
    - **if** $p_i \notin$ alive $\wedge$ $p_i \notin$ detected **then**
      - detected := detected $\cup$ { p_i }
      - **trigger** ⟨ crash p_i ⟩
    - **trigger** ⟨ pp2pSend p_i HEARTBEAT ⟩
  - alive := $\emptyset$; startTimer (TimeDelay)
- **upon event** ⟨ pp2pDeliver src, HEARTBEAT ⟩ **do**
  - alive := alive $\cup$ { src }

Eventually Perfect Failure Detector (◊P)



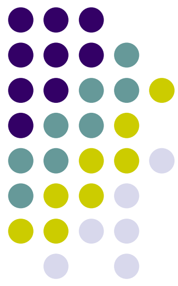
- ***Events***
  - Indication: ⟨suspect p⟩
  - Indication: ⟨restore p⟩
- ***Properties:***
  - Strong Completeness
  - Eventual Strong Accuracy: eventually no correct process is suspected by any correct process

Other typical Failure Detectors



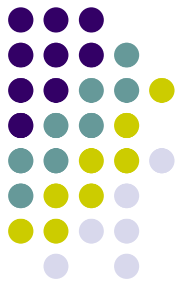
- Strong Detector (S)
  - Complete, Weakly Accurate
- *Weakly Accurate*
  - There exists a correct process which is never suspected by anyone
- For all failure patterns
  - For all possible behaviors of a detector
    - There exists a correct process P
      - All correct processes will never suspect P

Other typical Failure Detectors



- Eventually Strong Detector (◊S)
  - Complete,
  - Eventually Weakly Accurate P
- *Eventually Weakly Accurate*
  - After some finite time, t, the detector is weakly accurate

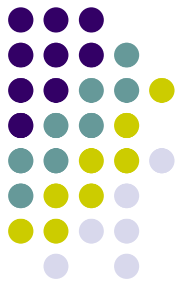
- After some time, the requirements are fulfilled
  - Prior to that, any behavior is possible!



Overview

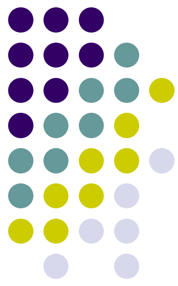
- (1) ***Why?*** Motivation
- (2) ***Where?*** Between the network and the application
- **(3)** ***How?*** (3.1) Specifications, (3.2) assumptions, and **(3.3)** algorithms

Combining Abstractions



- **Fail-stop**
  - Crash-stop process model
  - Perfect links + Perfect failure detector (P)
- **Fail-silent**
  - Crash-stop process model
  - Perfect links
- **Fail-noisy**
  - Crash-stop process model
  - Perfect links + Eventually Perfect failure detector (◊P)
- **Fail-recovery**
  - Crash-recovery process model
  - Stubborn links + …

Modules of a process



Applications

indication

request (deliver)

Algorithmic Modules

indication

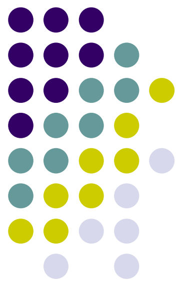
request (deliver)

indication

Channels

request (deliver)

Algorithms, events, happens before



p1

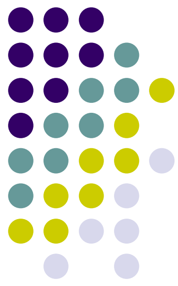
m3

m1

p2

m2

p3



Algorithms

p1

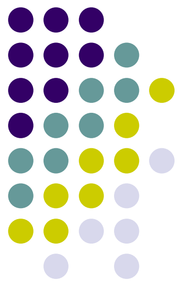
m1

p2

m2

p3

crash



The rest

- (A) We assume a crash-stop system with a perfect failure detector (fail-stop)
  - We give algorithms
- (B) We try to make a weaker assumption
  - We revisit the algorithms