Distributed algorithms

Prof R. Guerraoui
lpdwww.epfl.ch

Exam: Written
Reference: Book - Springer Verlag –
http://lpd.epfl.ch/site/education/da
- Introduction to Reliable (and Secure) Distributed Programming -
Algorithms (History)

- M. Al-Khawarizmi ~9th century: inventor of the zero, the decimal system, Arithmetic and Algebra

- A. Turing: all machines are equal
What is an algorithm?

- An ordered set of elementary instructions
- All execute on the same Turing machine
- Complexity measures the number of instructions (variables)
Distributed algorithms

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

We study algorithms for *distributed* systems.

A new way of thinking about algorithms and their complexity.

Whereas a centralized algorithm is the soul of a computer, a distributed algorithm is the soul of a *society* of computers.
Important

- This course is complementary to the course (concurrent algorithms)

- We study here *message passing* based algorithms whereas the other course focuses on *shared memory* based 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
A distributed system
Clients-server

Client A

Client B

Server
Multiple servers
(genuine distribution)
Applications

- Traffic control
- Reservation systems
- Banking
- Pretty much everything on the cloud
The optimistic view

Concurrency => speed (load-balancing)

Partial failures => high-availability
The pessimistic view

- Concurrency (interleaving) => incorrectness

- Partial failures => incorrectness
Distributed algorithms (Today: Google)

- Hundreds of thousands of machines connected
- A Google job involves 2000 machines
- 10 machines go down per day
Satoshi Nakamoto (2008) Nick Szabo

2009: 0.001 $

2016: 600 $
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 → Algorithms → Channels → Applications
Distributed systems

- The application needs underlying services for distributed interaction.
- The network is 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
- Leader election
- Terminating reliable broadcast
- ......
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 a common decision on whether to commit or abort a transaction
Underlying services

(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
Processes

A process executes a step at every tick of its local clock: a step consists of

- A local computation (local event) and message exchanges with other processes (global event)

NB. One message is delivered from/sent to a process per step
Processes

The program of a process is made of a finite set of modules (or components) organized as a software stack.

Modules within the same process interact by exchanging events.

upon event < Event1, att1, att2,..> do

// something

trigger < Event2, att1, att2,..>
Modules of a process

Applications

Algorithmic Modules

Channels

request (deliver) indication

request (deliver) indication

request (deliver) indication

request (deliver) indication
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 \(\sim\) liveness + safety

**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.

**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 a common 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
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 Assumptions on processes and channels

3.2.2 Failure detection
Processes

- A process either executes the algorithm assigned to it (steps) or fails
- Two kinds 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)

Many models are in between
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
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)
Processes/Channels

Processes communicate by message passing through communication channels

Messages are uniquely identified and the message identifier includes the sender’s identifier
Fair-loss links

**FL1. Fair-loss:** If a message is sent infinitely often by $pi$ to $pj$, and neither $pi$ or $pj$ crashes, then $m$ is delivered infinitely often by $pj$

**FL2. Finite duplication:** If a message $m$ is sent a finite number of times by $pi$ to $pj$, $m$ is delivered a finite number of times by $pj$

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

**SL1. Stubborn delivery:** if a process pi sends a message m to a correct process pj, and pi does not crash, then pj delivers m an infinite number of times

**SL2. No creation:** No message is delivered unless it was sent
Algorithm (sl)

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

upon event < sp2pSend, dest, m> do
  while (true) do
    trigger < flp2pSend, 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
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
    - trigger `< pp2pDeliver, src, m>`;
  - add m to delivered;
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
Overview

(1) \textbf{Why?} Motivation

(2) \textbf{Where?} Between the network and the application

(3) \textbf{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.
A failure detector module is defined by events and properties

**Events**

- Indication: \(<\text{crash}, \text{p}>\)

**Properties:**

- Completeness
- Accuracy
Failure Detection

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

Implementation:

1. Processes periodically send 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 doubles its time-out
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

**Eventually Synchronous:** the timing assumptions hold eventually

**Asynchronous:** no assumption
Overview

r (1) **Why?** Motivation

r (2) **Where?** Between the network and the application

r (3) **How?** (3.1) Specifications, (3.2) assumptions, and (3.3) algorithms
Algorithms
modules of a process

Applications

Algothmic Modules

Channels

request → indication
(deliver)

request → indication

request → indication
(deliver)

request → indication
(deliver)
Algorithms

p1

m1

p2

m2

p3

× crash
For every abstraction

- (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
Content of this course

- Reliable broadcast
- Causal order broadcast
- Shared memory
- Consensus
- Total order broadcast
- Atomic commit
- Leader election
- Terminating reliable broadcast

...