Synchronisation in Distributed Systems

Distributed Systems
Sistemi Distribuiti

Andrea Omicini
andrea.omicini@unibo.it

Dipartimento di Informatica – Scienza e Ingegneria (DISI)
Alma Mater Studiorum – Università di Bologna a Cesena

Academic Year 2013/2014

Outline

These Slides Contain Material from [Tanenbaum and van Steen, 2007]

Slides were made kindly available by the authors of the book

- Such slides shortly introduced the topics developed in the book [Tanenbaum and van Steen, 2007] adopted here as the main book of the course
- Some of the material from those slides has been re-used in the following, and integrated with new material according to the personal view of the teacher of this course
- Every problem or mistake contained in these slides, however, should be attributed to the sole responsibility of the teacher of this course

Outline

Communication & Interaction in Distributed System

Communication is just half of the story

- Interaction is a more general issue
- Governing (inter)action is a fundamental issue in (distributed) systems
- Doing the right thing at the right time is essential
- "At the right *time*" is the critical problem

Time in Distributed System

Synchronisation

- Is there a notion of time in a distributed system?
- Is there a notion of *global* time in a distributed system?
- If not, what can we do about this?
- How can we *synchronise* activities within a distributed system?

Outline

The Issue of Time

Time in distributed systems

- In centralised systems, time is unambiguous
- In a distributed system, there is not a *natural* notion of time
- Is it *possible* to build up a global notion of time in any distributed system?
- Is it *useful* to build up a global notion of time in any distributed system?

Physical Clocks I

Timers

- A *clock* in a computer is actually a *timer* – typically, an oscillating quartz with a *counter* and a *holding register*
- When the counter gets to zero, an interrupt is generated, and the counter is reloaded from the holding register
- Each interrupt is a *clock tick*

Physical Clocks II

Multiple CPUs

- No way to ensure two different crystals oscillate exactly at the same frequency
- Different clocks gradually get out of synch – *clock skew* is the difference in time
- Need for synchronising algorithms!
- Two approaches
  - global absolute time
  - global relative time

Global Absolute Time I

Absolute time

- Absolute time is handled by BIH (Bureau International de l'Heure) in Paris
- Expressed in terms of Universal Coordinated Time (UTC)
- Broadcasted as a short radio pulse (WWV) by NIST (National Institute of Standard Time) every UTC second, and by satellites providing UTC service
- If one machine in the system has access to an UTC service, an algorithm can be used that synchronises all machines based on this

Global Absolute Time II

Example: NTP

- Network Time Protocol (NTP)
- A time server has the global absolute time, and other machines have to synchronise
- Notice: clocks can only run forward – corrections cannot bring clocks backward

Global Relative Time

Relative time

- Sometimes, the only thing needed is that there is a shared time, regardless of absolute time
- So, algorithms based on active servers polling other servers to find out the average time, and the required estimated corrections as well
- No machine is required to have UTC time

Examples

- The Berkeley Algorithm: time daemons in all machines poll and respond to each other, and agree on a common time
- Reference Broadcast Synchronisation (RBS): global relative time in wireless networks

Outline

Physical vs. Logical Time

Physical time not always needed

- Till now, we have implicitly assumed that synchronisation is related to physical time
- However, we have also seen the case where the only need is a shared notion of time (a shared clock) among the processes of a distributed system, with no need for it to be exactly the "real" time
- As a step further, we may observe that often the only need for a distributed system is a shared clock, even *unrelated* to real time
- A notion of *logical time* is both possible and useful

Logical Clocks [Lamport, 1978]

Synchronisation is possible with no need to be absolute

- If two processes do not interact, there is no need of synchronisation—lack of synchronisation would not be observable
- Often, what really matters is not the exact time when events occur, but the *order* in which events occur
- Example: UNIX `make`

Logical clocks

- Synchronisation of non-physical, *logical clocks* is then both admissible and useful

Notation

Relation *happens-before*

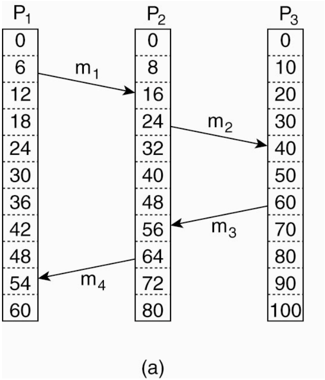
- $a \rightarrow b$ reads "a happens before b", and means that all processes agree that a occurs first, then b occurs
- $a \rightarrow b$ can be directly observed in two situations
  1. if a and b are events of the same process, and a comes before b, then $a \rightarrow b$ — local events are ordered by local time
  2. if a message is sent by process with an event a, and received by another process with an event b, then $a \rightarrow b$ — a message takes a finite, positive, non-zero amount of time to propagate from sender to receiver
- $a \rightarrow b$ is a transitive relation: $a \rightarrow b$, $b \rightarrow c$ imply $a \rightarrow c$
- *happens-before* defines a partial ordering over the events in a distributed system: when neither $a \rightarrow b$ nor $b \rightarrow a$ can be observed, then nothing can be said on their ordering — a and b are said to be *concurrent*

Logical Time

Measuring time with logical clocks: *time values*

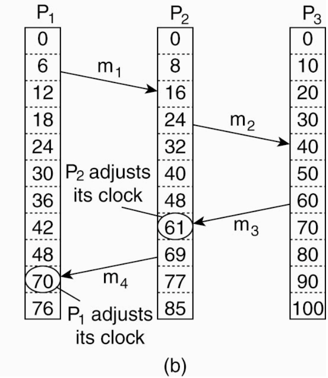
- A shared notion of time for an event a: *time value* $C(a)$ is such that every process agrees upon it
- Time value should be thought as the value of a logical clock upon which processes agree
- Time values are such that $a \to b$ implies $C(a) < C(b)$ — that is, time values should be assigned so that $C(a) < C(b)$
  1. if a and b are events of the same process, and a comes before b, then $C(a) < C(b)$
  2. if a message is sent by process with an event a, and received by another process with an event b, then $C(a) < C(b)$
- Since neither physical nor logical clocks can run backward, any correction to clock time should go forward (increasing), never backward (decreasing)

Lamport's Algorithm I



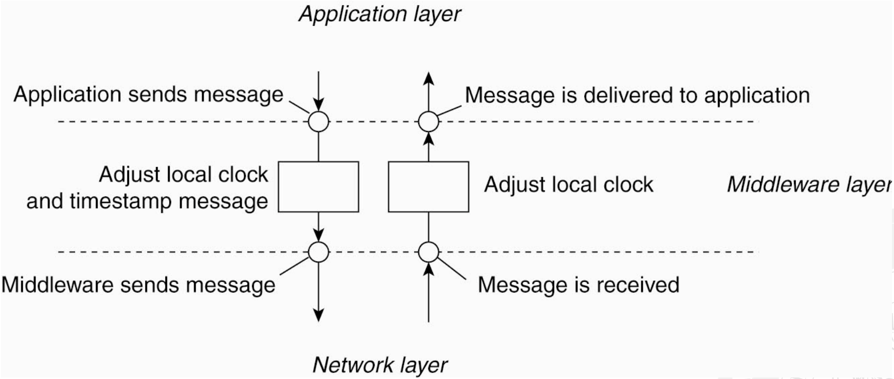
Concurrent message transmission using logical clocks
[Tanenbaum and van Steen, 2007]

Lamport's Algorithm II



Lamport's algorithm corrects the clocks
[Tanenbaum and van Steen, 2007]

Lamport's Algorithm III



Middleware support for Lamport's logical clocks
[Tanenbaum and van Steen, 2007]

Lamport's Algorithm IV

Implementation of Lamport's logical clocks

- Each process P_i maintains a *local* counter C_i
- Local counters are updated following three steps
  1. before executing an event, P_i executes $C_i \leftarrow C_i + 1$
  2. when sending a message m to P_j, process P_i sets m's timestamp $ts(m)$ to C_i after updating its counter (see step above)
  3. upon reception of a message m, process P_j adjusts its local counter such that $C_j \leftarrow max(C_j, ts(m))$, then goes back to step (1) and delivers the message to the application

! Sometimes, it is required that no two events occur exactly at the same time – process label can be added as a decimal number to the timestamp

Lamport's Algorithm V

Distributed implementation of global time

- C_i is local time at process P_i
- a is an event in a distributed system
- $\forall a \in P_i, C \leftarrow C_i(a)$

$\rightarrow$ C is the *global time* for the distributed system

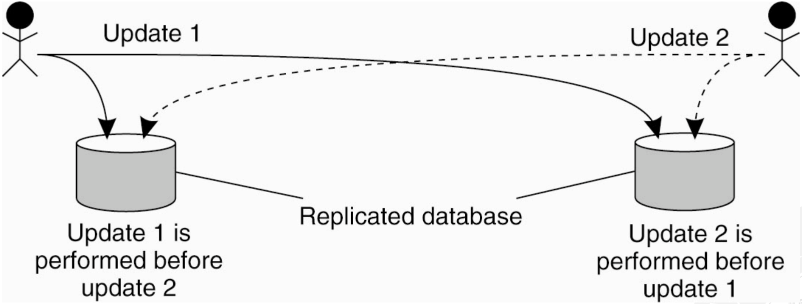
An Example I

Totally-ordered multicast

- A replicate database exists of the accounts of a bank in LA and NY
- A customer adds \$100 to his account, while at the same time a bank employee applies a 1% increment to the account
- Given that the original account contained \$1000, it may easily happens that, say, the LA replica records \$1110, the NY one \$1111

→ Inconsistency due to concurrent updates over a distributed replicated database

An Example II



Inconsistency in a replicated database after two concurrent updates [Tanenbaum and van Steen, 2007]

The Solution: Totally-ordered Multicast I

Assumptions

- A group of processes multicasting each other
- Each messaged is timestamped by the sender with its local logical time
- Also the sender conceptually receives the multicasted message
- Messages from the same sender are received in the same order they are sent, and no message is lost

The Solution: Totally-ordered Multicast II

Algorithm

- Each process maintains a local queue of all messages received, ordered according to its timestamp
- Every message received is acknowledge with a multicasted message, timestamped according to Lamport's algorithm

→ Timestamp of a received message is lower than the timestamp of the acks

→ Every process has essentially the same queue

- Only when all acknowledgements have been received, the middleware can deliver a queued message to the application
- Since all queues are equal, all messages are delivered to the application level at the same time on all the machines in the distributed system

The Solution: Totally-ordered Multicast III

Result

- A totally-ordered multicasting is perceived at the application level — as provided by the middleware layer
- In the example above, either the client or the employee command is issued first on all replicas

→ All replicas will be consistently updated

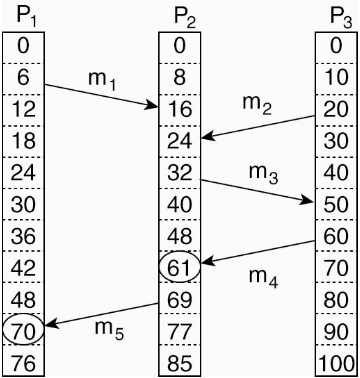
→ No idea, however, on whether the final record will be \$1110 or \$1111. . .

Vector Clocks I

The problem

- In essence, $a \rightarrow b$ implies $C(a) < C(b)$, whereas $C(a) < C(b)$ does not imply $a \rightarrow b$
  - so that, for instance, time values could be totally ordered when events are not
  - when events are unrelated, comparison of time values is meaningless
- Lamport's logical clocks say nothing about that
- Something more is needed
- To say in particular whether a and b are (un)related

Vector Clocks II



Concurrent message transmission using logical clocks [Tanenbaum and van Steen, 2007]

Vector Clocks III

Causality

- *m*1 is received before *m*2 is sent, according to Lamport's clock: can we conclude anything about *m*1 and *m*2?
- In general, the problem is that Lamport's clocks do not capture *causality*
- *Vector clocks* capture causality

Vector Clocks IV

Definition

- A vector clock $VC(a)$ assigned to an event a is such that $\exists b, VC(a) < VC(b) \longrightarrow a$ causally precedes b
- Each process P_i maintain a vector VC_i such that
  - $VC_i[i]$ is the number of events occurred so far at P_i — basically, the logical clock of P_i
  - $\leftarrow$ Every new event occurring in P_i increments $VC_i[i]$
  - $VC_i[j] = k$ means that P_i knows that k events have occurred at P_j — basically, the logical clock of P_j according to P_i's best knowledge

$\leftarrow$ Every message from P_i is timestamped with vector VC_i

Vector Clocks V

Algorithm

- Before any event is executed at P_i, $VC_i[i] \leftarrow VC_i[i] + 1$
- A message m from P_i to P_j timestamped with vector VC — $ts(m) = VC$
- A message m received by P_j makes it adjust VC_j such that $\forall k, VC_j[k] \leftarrow max(VC_j[k], ts(m)[k]$ — then m is delivered up to the application level

Vector Clocks VI

Result

- Every process knows how many events have preceded the sending of the received message at the sender process—information about the "chain of events" is preserved and shared among processes
- Each $ts(m)[i]$ refers to the events causally preceding m within each process P_i
- $ts(m)$ tells how many events may causally precede the sending of m, on which m may causally depend

← As a result, for instance, the delivery of a message to the application level could be suspended until all preceding messages from the same source are received

Enforcing Causal Communication

Causally-ordeded multicasting

- Using vector clocks, a message could be delivered only when all messages causally preceding it have been received
- ...assuming that all messages are multicasted in a group

! Weaker than totally-ordered multicasting: if two messages are not causally related, they could be delivered to applications in any order

Outline

Beyond Synchronisation I

Ordering events is not enough

- Sometimes, more articulated policies are required
- For instance, to ensure that concurrent accesses to a shared resource could harm its consistency, or corrupt it

Mutual exclusion

- A number of algorithms — centralised, decentralised, distributed — for instance, Token Ring
- We do not review them here
- The main point: some of them are based on a coordinator, all of them are *coordination* algorithms

Beyond Synchronisation II

Election algorithms

- Many distributed algorithms requires a *coordinator* to be elected
- Again, we do not review them: election algorithms are (used by) coordination algorithms

It is not merely a matter of time

- Synchronisation is about *when* things happen
- Actions are more than sending messages
- Interaction does not merely translate into suitably-ordered distributed actions — undifferentiated actions
- Actions have a nature, and meaningful interaction within a distributed system typically depends on such a nature

Beyond Synchronisation III

The problem of coordination

- Governing interaction based both on time, and on the nature of actions, and aimed at the achievement of some global objective for the distributed system
- This is the problem of *coordination*

Summing Up

Time in distributed systems

- The issue of time
- Physical time / clock
- Logical time / clock
- Causality and vector clocks

Toward coordination

- What do we do when we have some coherent notion of time?
- Coordinators and distributes algorithms

References I

Lamport, L. (1978).
Time, clocks, and the ordering of events in a distributed system.
Communications of the ACM, 21(7):558–565.

Tanenbaum, A. S. and van Steen, M. (2007).
Distributed Systems. Principles and Paradigms.
Pearson Prentice Hall, Upper Saddle River, NJ, USA, 2nd edition.

Synchronisation in Distributed Systems

Distributed Systems
Sistemi Distribuiti

Andrea Omicini
`andrea.omicini@unibo.it`

Dipartimento di Informatica – Scienza e Ingegneria (DISI)
Alma Mater Studiorum – Università di Bologna a Cesena

Academic Year 2013/2014