COMP2207: Distributed Systems and Networks - Time

Dr. Tope Omitola & Dr. Corina Cîrstea, University of Southampton © 2017
Contents

1. What is Time & Why Do We Need Time in Distributed Systems?

2. Physical Time

3. Clock skew and Clock drift in Distributed Clocks
4. Physical Clock Synchronisation - External and Internal Clock Synch. Algorithms

5. Causality & Logical Clocks

6. Vector Clocks
Introduction

Reading: Chapter 14 (Sections 14.1-14.4) of Coulouris et. al.

What is Time & Why Do We Need Time in Distributed Systems?

- Several reasons why Time is important & interesting in Distributed Systems

- Time is a quantity we often want to measure accurately
• In order to know at what time of day a particular event occurred at a particular computer it is necessary to synchronise its clock with an authoritative, external source of time (e.g. an eCommerce transaction involves events at your computer, a merchant’s computer and at a bank’s computer

• Wireless sensor networks rely on accurately synchronised clocks to compute the trajectory of fast-moving objects

• Consistency maintenance among replicated data relies on which update is the most recent one
• Versioning systems, e.g. git, need to know timestamps of source files in order to know compilation sequences

• Real-time systems, e.g. air-traffic control, must have accurate knowledge of time to provide useful service and avoid catastrophe

• High-frequency trading relies heavily on the accuracy of time synchronisation

• distinction between asynchronous and synchronous systems:
1. no timing assumptions can be made in asynchronous systems

2. bounds on message transmission delays or time taken to execute process steps can be assumed in synchronous systems
What is Time?

- Time is different in different cultures

- Solar calendars (the earth rotating around the sun), Lunar calendars (the earth rotating around the moon) and luni-solar calendars (a mixture of the two)

- What you use as your authoritative external reference source will determine how long is your day, week, month and year

Before addressing Time in distributed systems, let us look at Physical Time.
Physical Time

• According to the laws of physics and astronomy, real time is defined in terms of the rotation of Earth in the solar system.

• As we use the solar calendar; a solar second equals $1/86,400^{th}$ part of a solar day, which is the amount of time that the Earth takes to complete one revolution around its own axis.

• This is the primary standard of time also known as “Newtonian time”
Our watches are secondary standards of time and are calibrated with respect to the primary standard.

Modern timekeepers use atomic clocks as a de facto primary standard of time. A second is precisely the time for $9,192,631,770$ orbital transitions of the cesium 133 atom.

In actual practice, there is a slight discrepancy - $86,400$ atomic seconds is close to $3$ ms less than a solar day, so when the discrepancy grows to about $1$ s, a leap second is added to the atomic clock.
Sequential and Concurrent Events

- Despite technological advances, the clocks commonly available at the processors distributed across a system do not exactly show the same time.

Skew between computer clocks in a distributed system
• Built-in atomic clocks are not yet cost-effective e.g. wireless sensor networks cannot (yet) afford an atomic clock at each sensor node, though accurate time-keeping is crucial to detecting and tracking fast-moving objects

• And, the special theory of relativity tells us that simultaneity has no absolute meaning - it is relative to the location of the observers
Clock skew and Clock drift

- With no perfectly reliable timekeepers, 2 different physical clocks at 2 different locations will always drift

- **Skew** - the instantaneous difference between the readings of any two clocks

- Clocks can be synchronised; the accuracy of synchronisation depends on clock drift as well as on the resynchronisation interval, e.g. 6:00 p.m. for Bob is not necessarily exactly 6:00 p.m. for Alice
at a different location, even if they live in the same time zone

- The difference between 2 clocks might be extremely small, but the difference accumulated over many clock oscillations leads to an observable difference in the counters registered by the 2 clocks

- A clock’s drift rate is the change in the offset (difference in reading) between the clock and a nominal perfect reference clock per unit of time measured by the reference clock

- Computer clocks can be synchronised to external sources of highly accurate time. The most accurate
physical clocks use atomic oscillators, whose drift rate is about one part in $10^{13}$

One of these external sources of precise time is Coordinated Universal Time (UTC) abbreviated as UTC (from the French equivalent).

Another source of precise time is GPS. A system of 32 satellites deployed in the Earth’s orbit maintains accurate spatial coordinates and provides precise time reference almost everywhere on Earth where GPS signals can be received.
Physical Clock Synchronisation

- The availability of synchronised clocks is important, and simplifies many problems, in distributed systems.
- Consider a distributed system of $n$ physical clocks $(0, 1, 2, 3, \ldots, n−1)$ ticking approx. same rate.
- Clocks may not reflect real time & readings may drift apart.
- Mechanisms needed to periodically re-synchronise these clocks to within acceptable bounds.
• Bounded synchronised clocks are important in many applications, e.g.

• Air-traffic control systems rely on accurate time-keeping to monitor flight paths and avoid collisions

• Some security mechanisms depend on the physical times of events, so a loss of synchronisation may be a potential security lapse
Physical Clock Synchronisation (cont.)

Three Main Issues to Consider in Physical Clock Synchronisation

- **Bounded synchronisation**: full synchronisation not possible, but accuracy within a certain bound can be achieved

- **External synchronisation** (with external source of time $S$): $|S(t) - C_i(t)| < D$ for all $i$
  
  - can be achieved using time server
• Internal synchronisation: \(|C_i(t) - C_j(t)| < D\) for all \(i, j\)

  – can be achieved using internal coordinator (master)
Synchronisation in a Synchronous System

• assume bounds on message transmission delay and time to execute process steps

• to achieve internal synchronisation between two processes:
  – one process sends its time \( t \) to the process in a message
  – receiving process sets its time to \( t + (\max + \min)/2 \)
* $\max$ ($\min$) is the maximum (resp. minimum) time needed to transmit a message (Note in an asynchronous system, $\max$ is not known!)
External Synchronisation: Christian’s algorithm

- time server process supplies time according to its clock:

- process $p$ updates its time to $t + T_{round}/2$, where $T_{round}$ is the time taken by the request and reply.
• making several requests can improve the accuracy of the estimation

• using several time servers can improve availability and accuracy of estimation
Internal Synchronisation: Berkeley algorithm

- one machine chosen as *master*

- periodically, master polls other machines for their clock values

- master estimates local times based on round trip times and averages the values (including own time)
  - fault-tolerant average used to eliminate readings from faulty clocks
  - master clock set to new average
• master sends the amount by which processes need to adjust their individual times, and *not the new clock value!*
The Network Time Protocol (NTP)

- previous algorithms work well on intranets
- NTP allows clients across the Internet to be accurately synchronised to Universal Time
- NTP service provided by network of servers, organized in levels (primary servers, secondary servers etc)
– primary servers (1) connected directly to a time source (e.g. radio clock)

– secondary servers (2) synchronise with primary ones

– can use multicast or Christian’s algorithm for synchronisation at each level

– reconfiguration needed when failures occur
Causality

- Reminder: Above, we said: “Thus, 6:00 p.m. for Bob is not necessarily exactly 6:00 p.m. for Alice at a different location, even if they live in the same time zone”

- How do we decide if 2 events are concurrent? And, which one happens before another one?

- Causality is a basic issue in distributed computing - ordering of events fundamental and subtle

- Causal order is the basis of logical clocks
Logical Clock

- An event corresponds to the occurrence of an action.
- A set of events \((a, b, c, \ldots)\) in a single process is called sequential.
- Their occurrences can be **totally ordered** in time using the clock at that process.
- Bob returns home @ 5:40 pm, answers phone @ 5:50 pm, eats dinner @ 6:00 pm.
• Events \{ReturnHome, AnswerPhone, EatDinner\} define an ascending, sequential, total order.

• But 6 pm for Bob may not be exactly 6 pm for Alice. How do we settle such issues?
Causal Ordering

- Depend on Obvious Law of Nature: No message can be received before it is sent (Causality)

- If Bob sends message to Alice: Sending Event happens before Receiving Event
(Potential) Causal Ordering

- Define $a \rightarrow b$ (a **happened before** $b$) by:
  
  - if $a$ and $b$ are events in the same process and $a$ occurs before $b$, then $a \rightarrow b$
  
  - if $a$ is the event of message *send* from process A, and $b$ is the event of message *receive* by process B, then $a \rightarrow b$
  
  - if $a \rightarrow b$ and $b \rightarrow c$ then $a \rightarrow c$

- $\rightarrow$ is a partial order (reflexive, transitive, antisymmetric).
- $a, b$ are **concurrent** (written $a \parallel b$) if neither $a \rightarrow b$ nor $b \rightarrow a$
Logical Clocks

- A *logical clock* is an event counter that respects causal ordering.

- Consider the sequence of events in a single sequential process, each processing having a counter $LC$ for its logical clock.

- Initially, $\forall$ process, $LC = 0$. The occurrence of events correspond to the *ticks* of Logical Clock local to process.

- Each time an event takes place, $LC$ is incremented.
Implementing Logical Clocks

Logical clocks can be implemented using 3 simple rules:

- **LC1**: Each time a local event takes place, increment \( LC \) by 1

- **LC2**: When sending a message, append the value of \( LC \) to the message

- **LC3**: When receiving a message,
  \[ LC = 1 + \max(\text{local } LC, \text{message } LC) \]
  where local \( LC \) is the local value of \( LC \) and message \( LC \) is the \( LC \) value appended with the incoming message
This provides the following limited guarantee for a pair of events \( a \) and \( b \): \( a \rightarrow b \Rightarrow LC(a) < LC(b) \). (However, the converse is not true)

The figure (next page) illustrates Logical Clock mechanism using space-time view of events in a distributed system consisting of three processes \( P, Q, R \): the horizontal lines indicate the time-lines of the individual processes, and the diagonal lines represent the flow of messages between processes. Each event is tagged with its logical clock value.
We can see that the ffg. holds: \( b \to h \) since \((b \to c) \land (c \to g) \land (g \to h)\)
Quiz

• Which other property holds?
• We can see that the ffg. holds:

• \( a \rightarrow d \) since \((a \rightarrow b) \land (b \rightarrow c) \land (c \rightarrow d)\)

• It is impossible to determine any causal ordering between events \((a, e)\) since neither \((a \rightarrow e)\) holds nor \((e \rightarrow a)\) holds

• In such a case, we say the 2 events are concurrent, i.e. \((a \parallel e)\)
Lamport’s Logical Clocks

- used to capture the happened-before relation numerically

- each process $p_i$ keeps own **logical clock** $L_i$ (monotonically increasing process counter) and uses it to timestamp local events

- processes update logical clocks and transmit their values in messages:
- $L_i$ incremented before assigning a timestamp to an event.
- When $p_i$ sends message $m$, it timestamps it with current value of $L_i$ (after incrementing it), piggybacking $t = L_i$ on $m$.
- On receiving message $(m, t)$, $p_j$ computes $L_j = \max(L_j, t)$ before incrementing $L_j$ and timestamping the event $\text{receive}(m)$. 
Properties of Logical Clocks

- Write $L(a)$ for timestamp of event $a$ at whatever process it occurred.

- Then: $a \rightarrow b$ implies $L(a) \leq L(b)$.

- The converse is not true!!

  - e.g. $L(e) \leq L(b)$ but $b \parallel e$
Totally Ordered Logical Clocks

- distinct events in different processes may have identical timestamps

- achieve total order on events (all pairs of distinct events are ordered) by incorporating process identifiers into timestamps:

  - **global logical timestamp of event** $e$: $(T_i, i)$,

  where

  * $T_i$ - local timestamp

  * $i$ - process identifier
\( - \) define \((T_i, i) < (T_j, j)\) if

* either \(T_i < T_j\)
* or \(T_i = T_j\) and \(i < j\)
Vector Clocks

- logical clocks only give $a \rightarrow b$ implies $L(a) \leq L(b)$

- vector clocks also guarantee the converse!

- Primary goal of vector clocks is to detect causality

- vector clock for a system of $N$ processes: array of $N$ integers

- each process $p_i$ keeps its own vector clock $V_i$ and uses it to timestamp local events
• timestamps are attached to messages and updated upon receipt of messages
Vector Clocks (contd.)

• Let $\mathbb{E}$ denote the set of all events in a distributed system of $n$ processes $0\ldots n-1$

• Let $\mathcal{I}$ denote the set of non-negative integer vectors of size $n$,

• Then, vector clock, $VC$ is a mapping: $VC : \mathbb{E} \rightarrow \mathcal{I}$

• Let $a, b \in \mathbb{E}$, and we denote the $i^{th}$ element of $VC(a)$ by $VC_i(a)$
Vector Clocks (contd.)

- We define a partial order $<$ among vector clock values as follows: $VC(a) < VC(b)$ iff

  1. $\forall i : 0 \leq i \leq n - 1 : VC_i(a) \leq VC_i(b)$

  2. $\exists j : 0 \leq j \leq n - 1 : VC_j(a) \leq VC_j(b)$

- For a pair of events $a, b$, if neither $VC(a) < VC(b)$ nor $VC(b) < VC(a)$ holds, then the events are concurrent, i.e. $a || b$

- Vector clock implementation is required to satisfy the following condition: $a \rightarrow b \iff VC(a) < VC(b)$
Vector Clocks. Implementation

• Each process $i$ initialises its vector clock $VC[i]$ to $0,0,\ldots,0$ ($n$ components)

• Subsequently, each process follows the following 3 rules:

  1. Rule 1: Each local event at process $i$ increments the $i^{th}$ component of its latest clock value by 1 (i.e. $VC_i[i] = VC_i[i] + 1$)

  2. Rule 2: The sender of a message appends the vector clock value of the send event to every message that it sends
3. **Rule 3:** When process $j$ receives a message with a vector clock value $T$ from another process, it first increments the $j^{th}$ component of its own latest vector clock by 1 ($VC_j[i] = VC_j[i] + 1$), and then updates its vector clock as follows:

$$\forall k : 0 \leq k \leq n - 1 : VC_k[j] := \max(T_k, VC_k[j])$$
Vector Clocks. Example

The event with vector time stamp (2,1,0) is causally ordered before the event with the vector time stamp (2,1,4) but is concurrent with the event having time stamp (0,0,2).