Distributed System 1

Francesco Penasa

February 22, 2020

`18/02/2020`

Synchronization in Distributed Systems Time plays a fundamental role in many applications.

1. Synchronize (physical) clocks to approximate global time
2. Abstract time with logical clocks capturing (causal) ordering
3. Collect global state
4. Coordinate access to shared resources.

1 Synchronizing Physical Clocks

To guarantee synch

1. Maximum **clock drift** ρ is a costant
2. maximum allowed **clock skew** δ is a parameter
3. if two clocks drift in opposite directions, during a time inteval ΔT they accumulate a skew of $2\rho\Delta t$, resynch need $\delta/2\rho$ seconds.

Synchroniza all clocks against a single one (accuracy) **OR** synchronize all clocks among themselves (agreement). **Time monocity must be preserved** (we cannot step backward a clock).

1.1 Protocols

1. Server-based soution: periodically each client update himself with a server clock adding the offset θ. **Problem**: time might run **backwards** on the client machine, therefore, introduce change gradually; travel time of the message.
2. Network Time Protocol (NTP): protocol to synch time over large-scale networks `ntp.org`. The NTP servers are in Hierarchical structured (as DNS). Synchronization mechanisms: multicast (over LAN), procedure-call mode (server-based), symmetric mode(for higher levels; server exchange their roles and exchange the time).

1.2 Observation

1. Often is sufficient to agree on a time, even if it is not accurate.
2. what matters is ordering and causality (relative order). (example: allarm -¿ fire; fire -¿ allarm).
3. if there is no interaction, no synchronization is required.

2 Modeling a distributed execution

A distributed algorithm can be modeled as a collection of distributed automata.

relevant event in distributed algorithms

1. send event: *send(m,p)*
2. receive event: *receive(m)*
3. local event: everything else (set a variable, write a file...)

Histories

1. local history: history of a process p_i is a sequencec of events $h_i = e_i^0 e_i^1 \dots$
2. partial history (h_i^k): up to event e_i^k

Happens-before causality, it is useful only if we don't have global time $e \rightarrow e'$. Can be measured if e and e' are executed by the same process; e is the send and e' is the corresponding receive. Of course it is transitive. If we can not say happens before then it is concurrent.

Logical clocks enable coordination among processes without synchronization of physocal clocks, essetially a counter. **Definition:** Logical clock LC is a map function for the events e of the history H to an element of a time domain T

$$LC : H \rightarrow T$$

Clock consistency: events could be concurrent

$$e \rightarrow r' \Rightarrow LC(e) < LC(e')$$

Strong clock consistency events can not be concurrent

$$e \rightarrow e' \Leftrightarrow LC(e) < LC(e')$$

Scalar Clocks How to assign logical clocks in a way that guarantees **clock consistency**.
Definition Scalar logical clocks: an increasing counter, each process has its own logical clock, messages carry the timestamp
Update rule (+1 if is internal, max(mine,received + 1 if is receive)) which guarantees clock consistency by design.

Partial vs Total Order Scalar clocks achieve only partial ordering, to reach total ordering we add the process ID to clocks and we use a lexicographic order.

3 Examples

3.1 Totally Ordered Multicast

4 Questions

Scalar clock EXERCISE put the number on the events...

what are the problems on the server-based solutions?

1. **Minor:** There is already a mismatch for the "travel" time, to fix this we can use timestamps T1, T2, T3, T4 to measure the RTT.
2. **Major:** We could broke monotonicity of time, to fix the clock we should slow down until the time is fixed.

Definition of Happen-before We say that an event e happens-before an event e', and write $e \to e'$, if one of the following three cases is true:

$$\exists p_i \in \prod : e = e_i^r,\ e' = e_i^s,\ r < s$$

$$e = send(m, *) \wedge e' = receive(m)$$

$$\exists e'' : e \to e'' \to e'$$