Distributed Systems: Synchronization—Part I

Gian Pietro Picco
Dipartimento di Elettronica e Informazione
Politecnico di Milano, Italy
picco@elet.polimi.it
http://www.elet.polimi.it/~picco

Presented by:
Luca Mottola
mottola@elet.polimi.it
Synchronization in Distributed Systems

- The problem of synchronizing concurrent activities arises also in non-distributed systems
- However, distribution complicates matters:
  - Absence of a global physical clock
  - Absence of globally shared memory
  - Partial failures
- In these lectures, we study distributed algorithms for:
  - Synchronizing physical clocks
  - Simulating time using logical clocks & preserving event ordering
  - Leader election
  - Collecting global state & termination detection
  - Mutual exclusion
  - Distributed transactions
Time and Distributed Systems

- Time plays a fundamental role in many applications:
  - Execute a given action at a given time
  - Timestamping objects/data/messages enables reconstruction of event ordering
    - File versioning
    - Distributed debugging
    - Security algorithms
- Problem: ensure all machines “see” the same global time
Time

- Clocks vs. timers
- Time is a tricky issue per se:
  - Up to 1940, time is measured astronomically
    - 1 second = 1/86400th of a solar day
    - Earth is slowing down, making measures “inaccurate”
  - Since 1948, atomic clocks (International Atomic Time)
    - 1 second = 9,192,631,770 transitions of an atom of Cesium 133
    - Collected and averaged in Paris from 50 labs around the world
  - Skew between TAI and solar days accommodated by UTC (Universal Time Coordinates) when greater than 800ms
    - Greenwich Mean Time is only astronomical
    - About 30 leap seconds from 1958 to now
    - UTC disseminate via radio stations (DCF77 in Europe, WWV in US), GPS and GEOS satellite systems
Synchronizing Physical Clocks

• To guarantee synchronization:
  – Maximum \textit{clock drift} rate $\rho$ is a constant of the timer
    • For ordinary quartz crystals, $\rho=10^{-6}$ s/s, i.e., 1s every 11.6 days
  – Maximum allowed \textit{clock skew} $\delta$ is an engineering parameter
  – If two clocks are drifting in opposite directions, during a time interval $\Delta t$ they accumulate a skew of $2\rho \Delta t$
    $\Rightarrow$ resynch needed at least every $\delta/2\rho$ seconds

• The problem is either:
  – Synchronize all clocks against a single one, usually the one with external, accurate time information (\textit{accuracy})
  – Synchronize all clocks among themselves (\textit{agreement})

• At least time monotonicity must be preserved
• Several protocols have been devised
Simple Algorithms

- Cristian (1989)
  - Periodically, each client sends a request to the time server
  - Messages are assumed to travel fast w.r.t. required time accuracy
  - Problems:
    - Major: time might run backwards on client machine. Therefore, introduce change gradually (e.g., advance clock 9ms instead of 10ms on each clock tick)
    - Minor: it takes a non-zero amount of time to get the message to the server and back
      - measure round-trip time and adjust, e.g., $T_1 = C_{UTC} + T_{round}/2$
      - average over several measurements, ...

- Berkeley UNIX (1989)
  - The time server collects the time from all clients, averages, and then retransmits the required adjustment
Network Time Protocol (NTP)

- Designed for UTC synch over large-scale networks
  - Used in practice over the Internet, on top of UDP
  - Estimate of 10-20 million NTP clients and servers
  - Widely available (even under Windows)
  - ~1ms over LANs, 1-50ms over the Internet
  - More info at www.ntp.org

- Hierarchical synchronization subnet organized in strata
  - Servers in stratum 1 are directly connected to a UTC source
  - Lower strata (higher levels) provide more accurate information
  - Connections and strata membership change over time

- Synchronization mechanisms
  - Multicast (over LAN)
  - Procedure-call mode (similar to Cristian’s)
  - Symmetric mode (for higher levels)
Network Time Protocol (NTP)

- Each message bears timestamps of recent message events
- $T_{i-2} = T_{i-3} + t + o$ and $T_i = T_{i-1} + t' - o$
  - where $t$ and $t'$ are the messages’ transmission times
- A delay estimate is derived by imposing $d_i = t + t'$
- Similarly, the offset is $o_i = o + (t' - t)/2$
- The pairs $<o_i, d_i>$ are stored and processed through a statistical filter, to obtain a measure of reliability
- Each node communicates with several others
  - Only the most reliable data is returned to applications
  - Allows to select the best primary data source
Some Observations

- In many applications it is sufficient to agree on a time, even if it is not accurate w.r.t. the absolute time.
- What matters is often the ordering and causality relationships of events, rather than the timestamp itself.
- If two processes do not interact, it is not necessary that their clocks be synchronized.
Logical Time – Scalar Clocks


- Based on the **happens-before** relationship, \( A \rightarrow B \):
  - If events \( A \) and \( B \) occur in the same process and \( A \) occurs before \( B \), then \( A \rightarrow B \)
  - If \( A=send(msg) \) and \( B=recv(msg) \), then \( A \rightarrow B \)
  - \( \rightarrow \) is transitive

- If neither \( A \rightarrow B \) nor \( B \rightarrow A \), they are **concurrent** (\( A \parallel B \))

- Use integers to represent the clock value
  - No relationship with a physical clock whatsoever

\[ \begin{align*}
P1 & \quad A \quad B \\
P2 & \quad C \quad D \\
P3 & \quad E \quad F
\end{align*} \]
Assigning Scalar Clocks

- Each process keeps a logical scalar clock
  - Starts at zero
  - Increments clock upon each local event
  - Each message is timestamped with the sender’s scalar time
  - Upon receipt, the receiver’s clock is set to:
    \[ \text{MAX}(\text{msg timestamp}, \text{receiver’s clock}) + 1 \]

- Only partial ordering is achieved: total ordering can be obtained trivially by attaching process IDs to clocks
Example

(a)  

(b)
Example: Totally Ordered Multicast

- Updates in sequence:
  - Customer deposits $100
  - Bank adds 1% interest

- Updates are propagated to all locations:
  - If updates in the same order at each copy, consistent result (e.g., $1111)
  - If updates arrive in opposite orders, inconsistent result (e.g., $1110)

- Totally ordered multicast delivers messages in the same global order

- Using logical clocks:
  - Messages are acknowledged using multicast
  - All messages (including acks) carry a timestamp with the sender’s clock
  - Receivers store all messages in a queue, ordered according to its timestamp
  - Eventually, all processes have the same messages in the queue
    - Assumption: reliable and FIFO links
  - A message is delivered to the application only when it is at the top of the queue and all its acks have been received
Vector Clocks

- In scalar clocks, \( A \rightarrow B \Rightarrow C(A) < C(B) \)
  - But the reverse does not necessarily hold, e.g., if \( A \parallel B \)
- Scalar clocks do not capture all possible \textit{causality} relationships
- In vector clocks each process \( P_i \) maintains a vector \( V_i \) of \( N \) values (\( N=\#\text{processes} \)) such that:
  - \( V_i[i] \) is the number of events that have occurred at \( P_i \)
  - If \( V_i[j]=k \) then \( P_i \) knows that \( k \) events have occurred at \( P_j \)
- Rules for updating the vectors:
  - Initially, \( V_i[j]=0 \) for all \( i,j \)
  - Local event at \( P_i \) causes an increment of \( V_i[i] \)
  - \( P_i \) attaches a timestamp \( t=V_i \) in all messages it sends
  - When \( P_i \) receives a message containing \( t \), it sets \( V_i[j]=\max(V_i[j], t[j]) \) and then increments \( V_i[i] \)
Vector Clocks (cont’d)

- Relations on vector clocks
  - \( V(A) \leq V(B) \Leftrightarrow V_i(A) \leq V_i(B) \), for all \( i \)
  - \( V(A) < V(B) \Leftrightarrow V(A) \leq V(B) \) for all \( i \) and it exist a \( j \) such that \( V_i(A) < V_i(B) \)
  - \( V(A) \parallel V(B) \Leftrightarrow \neg V(A) < V(B) \land \neg V(B) < V(A) \)

- An isomorphism between the set of partially ordered events and their timestamps

- Determining causality:
  - \( A \rightarrow B \Leftrightarrow V(A) < V(B) \)
  - \( A \parallel B \Leftrightarrow V(A) \parallel V(B) \)
Examples

- By looking only at the timestamps we are able to determine whether two events are causally related or concurrent.
- Can be exploited to implement causal message delivery.
Example: Bulletin Boards

- Need to preserve the ordering **only** between messages and replies
- Totally ordered multicast is too strong
  - If M1 arrives before M2, it does not necessarily mean that the two are related
- Using vector clocks:
  - Increment clock only when sending a message
  - Hold a reply until the previous messages are received:
    - $ts(r)[j] = V_k[j] + 1$
    - $ts(r)[i] \leq V_k[i]$ for all $i \neq j$
Mutual Exclusion

• Required to prevent interference and ensure consistency of resource access
• Critical section problem, typical of OS
  – But here, no shared memory
• Requirements:
  – Safety property: at most one process may execute in the critical section at a time
  – Liveness property: all requests to enter/exit the critical section eventually succeed
• Simplest solution: a server coordinating access
  – Emulates a centralized solution
  – Server manages the lock using a “token”
  – Resource access request and release obtained with respective messages to the coordinator
  – Easy to guarantee mutual exclusion and fairness
  – Drawbacks: performance bottleneck and single point of failure
Mutual Exclusion with Scalar Clocks

- To request access to a resource:
  - A process $P_i$ sends a resource request message $m$, with timestamp $T_m$, to all processes (including itself)
  - The request is put into a local queue ordered according to $T_m$ (process ids are used to break ties)
  - Upon receipt of $m$, a process sends an acknowledgment to $P_i$, with timestamp $>T_m$

- To release a resource
  - A message is sent to everybody
  - When the release message is received, the request is removed from the queue

- A resource is granted to $P_i$ when
  - Its request message is ahead of all others in the request queue
  - Its request has been acknowledged by all the other processes with a message $>T_m$
Example

```
<table>
<thead>
<tr>
<th>Acquire</th>
<th>Request</th>
<th>Ack</th>
<th>Release</th>
</tr>
</thead>
</table>
```

Diagram:

- P1:1
- P2:1
- P3:1

Timeline:

1. Acquire
2. Request
3. Ack
4. Release
A Token Ring Solution

- Processes are logically arranged in a ring, regardless of their physical connectivity
  - At least for the purpose of mutual exclusion
- Access is granted by a token that is forwarded along a given direction on the ring
  - Resource access is achieved by retaining the token
  - Resource release is achieved by forwarding the token
- The token must circulate regardless of the existence of access requests

(a)

(b)
## Comparison

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Messages per entry/exit</th>
<th>Delay before entry (in message times)</th>
<th>Problems</th>
</tr>
</thead>
<tbody>
<tr>
<td>Centralized</td>
<td>3</td>
<td>2</td>
<td>Coordinator crash</td>
</tr>
<tr>
<td>Distributed (Lamport)</td>
<td>3 ( n – 1 )</td>
<td>2 ( n – 1 )</td>
<td>Crash of any process</td>
</tr>
<tr>
<td>Token ring</td>
<td>1 to $\infty$</td>
<td>0 to n – 1</td>
<td>Lost token, process crash</td>
</tr>
</tbody>
</table>

If nobody wants to enter the critical section, the token circulates indefinitely.
Leader Election

• Many distributed algorithms require a process to act as a coordinator (or some other special role)
  – E.g., for server-based mutual exclusion

• Problem: make everybody agree on a new leader
  – When the old is no longer available, e.g., because of failure or applicative reasons

• Minimal assumption: nodes are distinguishable
  – Otherwise, no way to perform selection
  – Typically use the identifier: the process with the highest ID becomes the leader

• Also, closed system: processes know each other
  – But do not know who is up and who has failed

• Algorithms differ on the selection process
Bully Election Algorithm

- The leader, 7, fails
  - Process 4 (noticing the failure of 7) holds an election
  - Process 5 and 6 respond, telling 4 to stop
  - Now 5 and 6 each hold an election
  - 6 tells 5 to stop
  - 6 wins and tells everyone

- When a process comes back (or joins) it holds an election
A Ring-Based Algorithm

- Assume a (physical or logical) ring topology among nodes
- When a process detects a leader failure, it sends an ELECTION message containing its ID to the closest alive neighbor
- Upon receipt of the election message on process $P$:
  - If $P$ is not in the message, add $P$ and propagate to next neighbor
  - If $P$ is in the list, change message type to COORDINATOR, and re-circulate
- On receiving a COORDINATOR message, a node considers the process with the highest id as the new leader (and is also informed about the remaining members of the ring)
- Multiple messages may circulate at the same time
  - Eventually converge to the same content