Synchronisation in Distributed Systems

Naming for searching communication partners

Communication Mechanisms for the communication process

But... Not enough for co-operation:
- Synchronisation
  - Atomic operations
  - Deadlock avoidance
  - Consistency in transaction processing
  - Simultaneous access to a shared resource
  - Ordering of events
  - ...

More complicated problems than in central systems!

Kinds of Synchronisation

- Synchronisation based on actual (absolute) time
- Synchronisation by relative ordering of events
- Distributed global states
- Using a coordinator: election mechanisms
- Mutual exclusion for protection against multiple access
- Distributed transactions

Clock Synchronization

- Clocks in distributed systems are independent
- Some (or even all) clocks are inaccurate
- When each machine has its own clock, an event that occurred after another event may nevertheless be assigned an earlier time.
- How to determine the right sequence of events?

Example Compiler - synchronisation is needed considering the absolute time on all machines:

```
Computer on which compiler runs: 2144 2145 2146 2147
<table>
<thead>
<tr>
<th>time</th>
<th>output.c created</th>
</tr>
</thead>
<tbody>
<tr>
<td>2144</td>
<td></td>
</tr>
<tr>
<td>2145</td>
<td></td>
</tr>
<tr>
<td>2146</td>
<td></td>
</tr>
<tr>
<td>2147</td>
<td></td>
</tr>
</tbody>
</table>
```

```
Computer on which editor runs: 2142 2143 2144 2145
<table>
<thead>
<tr>
<th>time</th>
<th>output.c created</th>
</tr>
</thead>
<tbody>
<tr>
<td>2142</td>
<td></td>
</tr>
<tr>
<td>2143</td>
<td></td>
</tr>
<tr>
<td>2144</td>
<td></td>
</tr>
<tr>
<td>2145</td>
<td></td>
</tr>
</tbody>
</table>
```

How can we - synchronise clocks with real world?
- synchronise clocks with each other?
Clocks

Necessary for synchronisation: assign a *timestamp* with each event

But... how to determine the own resp. all other times in the system?

- **Skew**: the difference between the times on two clocks (at any instant)
- Computer clocks are subject to *clock drift* (they count time at different rates)
- Clock *drift rate*: the difference per unit of time from some ideal reference clock
- Ordinary quartz clocks drift by about 1 sec in 11-12 days. (10^-6 secs/sec).
- High precision quartz clocks drift rate is about 10^-7 or 10^-8 secs/sec

**Universal Coordinated Time (UCT)**

- International Atomic Time is based on very accurate physical clocks (drift rate 10^-13)
- UTC is an international standard for time keeping
- It is based on atomic time, but occasionally adjusted to astronomical time
- It is broadcast from radio stations on land and satellite (e.g. GPS)
- Computers with receivers can synchronize their clocks with these timing signals (*But: only a small fraction of all computers have such receivers!*)
- Signals from land-based stations are accurate to about 0.1-10 milliseconds
- Signals from GPS are accurate to about 1 microsecond

**Clock Synchronization Algorithms**

- Universal Coordinated Time (as reference time): \( t \)
- Clock time on machine \( p \): \( C_p(t) \)
- Perfect world: \( C_p(t) = t \), i.e. \( dC_p / dt = 1 \)
  \[ \Rightarrow \text{Reality: there is a clock drift so that for a given } \rho : 1 - \rho \leq dC_p / dt \leq 1 + \rho \]
- To hold a maximum time drift \( \delta \), a re-synchronisation has to be made in certain intervals: all \( \delta / 2 \rho \) seconds
- Relation between clock time and UTC when clocks tick at different rates:

\[
\begin{align*}
\frac{dC}{dt} &> 1 \\
\frac{dC}{dt} &< 1 \\
\frac{dC}{dt} &= 1
\end{align*}
\]

\[ \text{UTC}, t \]

**Cristian's Algorithm**

- There is one *time server* \( T \) with a UCT receiver
- All other machines \( M \) are asking the time server at least all \( \delta / 2 \rho \) seconds
- \( T \) responds as fast as it can

\( M \) computes current time:
- Hold time \( t_{\text{send}} \) for sending the request
- Measure time till response with \( t_{\text{UTC}} \) arrives
- Subtract service time \( t_{\text{response}} \) of \( T \)
- Divide by two to consider only the time since the reply was sent
- Add ‘delivery time’ to the time \( t_{\text{sent}} \) sent by \( T \)
- Result \( t_{\text{synchronous}} \) becomes new system time

\[ t_{\text{synchronous}} = t_{\text{UTC}} + \frac{t_{\text{send}} - t_{\text{send}} - t_{\text{response}}}{2} \]

Both values are measured with the same clock

Consider message run-time, avoid \( M \)'s time to be moved back
The Berkeley Algorithm

Another approach (Berkeley Unix):

1. time server sends its time to all machines
2. the machines answer with their current deviation from the time server
3. the time server sums up all deviations and divides by the number of machines (including itself!)
4. the new time for each machine is given by the mean time

Important: fast clocks are not moved back, but instructed to move slower

Simple mechanism for decentralised synchronisation:

- divide time into fixed-length synchronisation intervals
- at the beginning of each interval all machines broadcast their current time
- start a timer to collect all values of other machines arriving in a given interval
- compute the new time
  - by simply averaging all answers, or
  - by discarding the m highest and the m lowest answers (to protect against faulty clocks), or
  - by averaging values corrected by an estimation of their propagation time.

...but: in large-scale networks, the broadcasting could become a problem

widely used algorithm in the Internet: Network Time Protocol (NTP)

Network Time Protocol (NTP)

NTP is a time service designed for the Internet

- **Reliability** from redundant paths
- **Scalable** to large number of clients and servers
- **Authenticates** time sources to protect against wrong time data
- NTP is provided by a network of servers distributed across the Internet
- Hierarchical structure:
  - Primary servers are connected to UTC sources
  - Secondary servers are synchronized to primary servers (Synchronisation subnet)
  - Lowest level servers in users' computers

More accurate time

Modes of synchronisation:

- **Multicast**
  - A server within a high speed LAN multicasts time to others which set clocks assuming some delay (not very accurate)
- **Procedure call**
  - A server accepts requests from other computers (like in Cristian's algorithm). Higher accuracy than using multicast (and a solution if no multicast is supported)
- **Symmetric**
  - Pairs of servers exchange messages containing time information
  - Used where very high accuracies are needed (e.g. for higher levels)

All modes use UDP to transfer time data
Messages exchanged between a pair of NTP peers

- Each message bears timestamps of recent events:
  - Local times of Send and Receive of previous message
  - Local times of Send of current message
- Recipient notes the time of receipt \( T_i \) (we have \( T_{i-3}, T_{i-2}, T_{i-1}, T_i \))
- In symmetric mode there can be a non-negligible delay between messages

\[
T_{i-2} = T_{i-3} + t + \alpha \quad \text{and} \quad T_i = T_{i-1} + t - \alpha
\]

This gives us (by adding the equations):

\[
d_i = t + t' = T_{i-2} - T_{i-3} + T_i - T_{i-1}
\]

Also (by subtracting the equations)

\[
\alpha = \alpha_i + (t' - t)/2 \quad \text{where} \quad \alpha_i = (T_{i-2} - T_{i-3} + T_i - T_{i-1})/2
\]

Using the fact that \( t, t' > 0 \) it can be shown that

\[
\alpha_i - d_i/2 \leq \alpha \leq \alpha_i + d_i/2
\]

Thus \( \alpha_i \) is an estimate of the offset and \( d_i \) is a measure of the accuracy

- NTP servers filter pairs \( <\alpha_i, d_i> \), estimating reliability from variation, allowing them to select peers
- Accuracy of 10s of milliseconds over Internet paths (1 on LANs)

Accuracy of NTP

- For each pair of messages between two servers, NTP estimates an offset \( \alpha \), between the two clocks and a delay \( d_i \) (total time for the two messages, which take \( t \) and \( t' \))

\[
T_{i-2} = T_{i-3} + t + \alpha \quad \text{and} \quad T_i = T_{i-1} + t - \alpha
\]

- This gives us (by adding the equations):

\[
d_i = t + t' = T_{i-2} - T_{i-3} + T_i - T_{i-1}
\]

- Also (by subtracting the equations)

\[
\alpha = \alpha_i + (t' - t)/2 \quad \text{where} \quad \alpha_i = (T_{i-2} - T_{i-3} + T_i - T_{i-1})/2
\]

Using the fact that \( t, t' > 0 \) it can be shown that

\[
\alpha_i - d_i/2 \leq \alpha \leq \alpha_i + d_i/2
\]

Thus \( \alpha_i \) is an estimate of the offset and \( d_i \) is a measure of the accuracy

- NTP servers filter pairs \( <\alpha_i, d_i> \), estimating reliability from variation, allowing them to select peers
- Accuracy of 10s of milliseconds over Internet paths (1 on LANs)

Lamport Timestamps

The absolute time is not needed in each case. Often enough:
ordering events only with respect to logical clocks

Relation: \( a \rightarrow b \) means that 'a happens before b'
(Meaning: all processes agree that event a happens before event b)

1. \( a \rightarrow b \) is true, when both events occur in the same process
2. \( a \rightarrow b \) is true, if one process is sending a message (event a) and another process is receiving this message (event b)
3. \( \rightarrow \) is transitive
4. neither \( a \rightarrow b \) nor \( b \rightarrow a \) is true, if they occur in two processes which do not exchange messages (Concurrent Processes/Events, notation: a|b)

Needed: assigning a value \( C(a) \) to an event a on which all processes agree, with \( C(a) < C(b) \) if \( a \rightarrow b \)

Lamport's Algorithm

Solution using the 'happens before' relation:

- sending local time with each message
- arriving before sending violates the 'happens before' relation
- in this case, forward the clock of the receiver to the next higher value

Addition: for all events a and b holds \( C(a) \neq C(b) \).
This can be achieved by attaching the local process numbers to the local time (eg. 1505200216053022.1300)
Application of Lamport Timestamps

Replicated database: updates have to be performed in a certain order.

Using Lamport's Timestamps:
- Each message is time stamped with the current (logical) time of the sender.
- The messages are sent to all receivers (and to the sender itself!).
- Received messages are ordered by their timestamps.
- Receivers multicast acknowledgements.
- Only after receiving acknowledgements from all receivers, the message with the lowest timestamp is read by the processes.

Enhancement: Vector Timestamps

Problem with Lamport timestamps: they do not capture causality.

Using vector timestamps:
- A vector timestamp VT(a) for event a is in relation VT(a) < VT(b) to event b, if a is known to causally precede b.

VT is constructed by each process as a vector V with:
1. V[i][j] is the number of events that have occurred so far at P_i.
2. If V[i][j] = k then P_i knows that k events have occurred at P_j.

When P_i sends a message m, then it sends along its current V_i. This timestamp tells the receiver P_j how many events in other processes have preceded m.

P_j adjusts its own vector for each k to V[j][k] = max{V[j][k], V_i[k]} (These entries reflect the number of messages that P_j has to receive to have at least seen the same messages that preceded the sending of m).

Add 1 to each entry for the event of receiving m.

Example:
Vector clock V_i at process p_i is an array of N integers.
- VC1: Initially V[i][j] = 0 for i, j = 1, 2, ..., N.
- VC2: before p_i timestamps an event it sets V[i][i] := V[i][i] + 1.
- VC3: p_i piggybacks t = V_i on every message it sends.
- VC4: when p_i receives (m, t) it sets V[j][j] := max(V[j][j], t[j]) (then before next event adds 1 to own element using VC2).

Global State

Often required: not only ordering of events, but global state of a distributed system.

Global state = local state of each process + messages currently in transit.

Examples:
- a. Garbage collection:
  - Object o seems to be garbage, but it has sent a message containing a reference to it.
- b. Deadlock:
  - Both processes are waiting for a message from the other process.
- c. Termination:
  - Both processes are passive and seem to be terminated, but in fact there is a message sent by p_2 to activate p_1.
Global State

- Graphically for global state: cut
  
  ![Consistent cut](image1)
  ![Inconsistent cut](image2)

  - Allows sent messages
  - Allows no received but not sent messages

  (a) Consistent cut
  (b) Inconsistent cut

  Chandy/Lamport: **distributed snapshot** (reflects a consistent global state)

Distributed Snapshot

**Taking a snapshot:**
- Any process $P$ can initialise the computation by recording the local state.
- $P$ sends a marker to each process to which he has a communication channel.
- $Q$ receives marker
  - First marker received $\Rightarrow$ record local state and send a marker on each outgoing channel.
  - All other markers: record all incoming messages for each channel.
  - One marker for each incoming channel received: stop recording and send results to $P$.

**Marker receiving rule for process $p_i$:**
- On $p_i$'s receipt of a marker message over channel $c$:
  - if ($p_i$ has not yet recorded its state) it
    - records its process state now;
    - records the state of $c$ as the empty set;
    - turns on recording of messages arriving over other incoming channels;
  - else
    - $p_i$ records the state of $c$ as the set of messages it has received over $c$ since it saved its state,
end if

**Marker sending rule for process $p_i$:**
- After $p_i$ has recorded its state, for each outgoing channel $c$:
  - $p_i$ sends one marker message over $c$
  (before it sends any other message over $c$).

Chandy and Lamport’s ‘snapshot’ algorithm