Synchronisation in Distributed Systems

Co-operation and Co-ordination in Distributed Systems

Naming for searching communication partners
Communication Mechanisms for the communication between processes

But... Not enough for co-operation:

- Synchronisation
- Atomic operations
- Deadlock avoidance
- Consistency in transaction processing

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

The Role of Time

- A distributed system consists of a number of processes.
- Each process has a state (values of variables)
- Each process takes actions to change its state, or to communicate with other processes (send, receive)
- An event is the occurrence of an action
- Events within a process can be ordered by the time of occurrence
- In distributed systems, also the time order of events on different machines and between different processes has to be known
- Needed: concept of “global time”, i.e. local clocks of machines have to be synchronised
Clock Synchronisation

- 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:

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?

Network

- Skew: the difference between the times on two clocks (at any instant)
- Computer clocks are subject to clock drift (they count time at different speeds)
- 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 atomic clocks (drift rate 10^{-13}). Problem: “Atomic day” is 3 msec shorter than a solar day
- UTC is an international standard for time keeping solving this problem
- It is based on atomic time, but occasionally adjusted to astronomical time: when the difference to the solar time grows up 800 msec, an additional leap second is inserted
- It is broadcasted from radio stations on land and satellite (e.g. GPS)
- Computers with receivers can synchronise their clocks with these timing signals (But: only a small fraction of all computers have such receivers!)
- Problem with received UTC: propagation delay has to be considered
  - Signals from land-based stations are accurate to about 0.1-10 milliseconds
  - Signals from GPS are accurate to about 1 microsecond

Clock Synchronisation Algorithms

- Universal Coordinated Time (as reference time): \( t \)
- Clock time on machine \( p \): \( C_p(t) \)
- Perfect world: \( C_p(t) = t \), i.e. \( \frac{dC}{dt} = 1 \)
  \( \Rightarrow \) Reality: there is a clock drift so that a maximum drift rate can be specified: \( \rho: 1 - \rho \leq \frac{dC}{dt} \leq 1 + \rho \)
- Needed for synchronisation: definition of a tolerable difference in clock values, the maximum time drift \( \delta \)
- With this, re-synchronisation has to be made in certain intervals: all \( \frac{\delta 2 \rho}{2} \) seconds

How to make such a re-synchronisation?
Cristian’s Algorithm

- There is one central time server $T$ with a UCT receiver.
- All other machines $M$ are contacting the time server at least all $\delta/2\rho$ seconds.
- $T$ responds as fast as it can.

$M$ computes current time:

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

$$t_{synchronous} = \frac{t_{send} + t_{receive} - t_{send} - t_{response}}{2}$$

Consider message run-time, avoid $M$’s time to be moved back.

The Berkeley Algorithm

- Another approach (Berkeley Unix):
  - active time server
  - logical synchronisation

1. The 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.

Distributed Algorithms

Problem with Cristian/Berkeley: use of a centralised server; mainly used in Intranets.
Simple mechanism for decentralised synchronisation (based on Berkeley Algorithm):
- divide time into fixed-length synchronisation intervals
- at the beginning of each interval all machines
  - broadcast their current time
  - collect all values of other machines arriving in a given time span
  - compute the new time
    - by simply averaging all answers, or
    - by discarding the $m$ highest and the $m$ lowest answers before averaging (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 by using 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 time servers distributed across the Internet

Hierarchical structure: synchronisation subnet tree

Primary servers are connected to UTC sources.
Secondary servers are synchronised to primary servers.
Lowest level servers in users’ computers, synchronised to secondary servers.

Note: this is only an example, there can be more than three layers.
NTP - synchronisation of servers

- The synchronisation subnet can reconfigure if failures occur, e.g.
  - a primary that loses its UTC source can become a secondary
  - a secondary that loses its primary can use another primary

- 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 Csiatia'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

- UTC is sent in messages between the servers
- Each message bears timestamps of recent events:
  - Local times of Send ($T_{i3}$) and Receive ($T_{i}$) of previous message $m$
  - Local time of Send ($T_{i}$) of current message $m'$
- Recipient of $m'$ notes the time of receipt $T_i$ (it then knows $T_{i3}$, $T_{i2}$, $T_{i1}$, $T_i$)
- In symmetric mode there can be a non-negligible delay between messages

Accuracy of NTP

- For each pair of messages between two servers, NTP estimates
  - an offset $o_i$ between the two clocks and
  - a delay $d_i$ (total time for transmitting the two messages, which take $t$ and $t'$).
- You have: $T_{i2} = T_{i3} + t + o_i$ and $T_i = T_{i1} + t' - o_i$
  - for the current offset $o_i$ between A and B

- This gives us (by adding the equations):
  $$d_i = t + t' = T_{i2} - T_{i3} + T_i - T_{i1}$$

- Also (by subtracting the equations)
  $$o_i = o_i - (t' - t)/2$$
  where $o_i = (T_{i2} - T_{i3} - T_i + T_{i1})/2$
- Using the fact that $t, t' > 0$ it can be shown that
  $$o_i - d_i/2 \leq o_i \leq o_i + d_i/2 .$$

- Thus $o_i$ is an estimate of the offset and $d_i$ is a measure of the accuracy
- NTP servers filter pairs $<o_i, d_i>$, estimating reliability of time servers from
  variations in pairs and accuracy of estimations by low delays $d_i$, allowing
  them to select peers
- Accuracy of 10s of milliseconds over Internet paths, 1 millisecond on LANs
Lamport Timestamps

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

**Relation:** happens-before: \( a \rightarrow b \) 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

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

### Lamport’s Algorithm

**Process 1**

- \( 0 \)
- \( 6 \)
- \( 12 \)
- \( 18 \)
- \( 24 \)
- \( 30 \)
- \( 36 \)
- \( 42 \)
- \( 48 \)
- \( 54 \)
- \( 60 \)

**Process 2**

- \( 8 \)
- \( 16 \)
- \( 24 \)
- \( 32 \)
- \( 40 \)
- \( 48 \)
- \( 56 \)
- \( 64 \)
- \( 72 \)
- \( 80 \)
- \( 88 \)

**Process 3**

- \( 10 \)
- \( 20 \)
- \( 30 \)
- \( 40 \)
- \( 50 \)
- \( 60 \)
- \( 70 \)
- \( 80 \)

**Solution using the ‘happens before’ relation:**

- initialise all clocks with 0
- sending local time with \( \rightarrow \) means that \( a \) happens before \( b \)
- ordering of events only with respect to logical clocks
- arriving before sending violates the ‘happens before’ relation. In this case, forward the clock of the receiver to the next higher value
- \( \rightarrow \) is true, when both events occur in the same process
- \( \rightarrow \) is true, if they occur in two processes which do not exchange messages
- \( \rightarrow \) is transitive
- neither \( a \rightarrow b \) nor \( b \rightarrow a \) is true, if they occur in two processes which do not exchange messages

### Application of Lamport Timestamps

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

- Update 1
- Update 2

**Required:** totally-ordered multicast.

**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**

**Definition:**

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 \( P_i \) as a vector \( V_i \) 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 \( P_i \)

**Using vector timestamps**

- When \( P_i \) sends a message \( m \), then it sends along its current \( V_i \)
- This timestamp vector tells the receiver \( P_j \) how many events in other processes have preceded \( m \)
- \( P_i \) adjusts its own vector for each \( k \) to \( V_i[k] = \max(V_i[k], V[j][k]) \) (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 entry \( V_i[j] \) for the event of receiving \( m \)
Vector Timestamps - Example

- Vector clock $V_i$ at process $p_i$ is an array of $N$ integers
- Initially $V_i[j] = 0$ for $i, j = 1, 2, \ldots N$
- Before $p_i$ timestamps an event it sets $V_i[j] := V_i[j] + 1$
- $p_i$ piggybacks $t = V_i$ on every message it sends
- When $p_i$ receives $(m, t)$ it sets $V_i[j] := \max(V_i[j], t[j])$ for $j = 1, 2, \ldots N$

Causality Violation

Vector timestamps can be used for detecting causality violations:

- Causality violation occurs when the order of messages causes an action based on information that another host has not yet received.
- In designing a distributed system, potential for causality violation is important.
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 p2 to activate p1

Problem with getting a global state: there is no global time!

➢ To do: get a global state from lots of local states recorded at different real times

➢ Graphically for global state: cut

A global state is consistent if it corresponds to a consistent cut

Distributed Snapshot

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

Assumptions:

• No process or communication failures occur, all messages arrive intact, exactly once
• Communication channels are unidirectional and FIFO-ordered
• There is a communication path between any two processes
• Any process may initiate the snapshot (sends Marker)
• Snapshot does not interfere with normal execution
• Each process records its state and the state of its incoming channels (no central collection)

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 ⇒ 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
Chandy and Lamport's 'snapshot' algorithm

Marker receiving rule for process pi:
On pi's receipt of a marker message over channel c:
- if (pi 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
- pi 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 pi:
After pi has recorded its state, for each outgoing channel c:
- pi sends one marker message over c (before it sends any other message over c).

Snapshot Example

1. P1 initiates snapshot: records its state (S1); sends Markers to P2 & P3; turns on recording for channels C21 and C31
2. P2 receives Marker over C12, records its state (S2), sets state(C12) = {}; sends Marker to P1 & P3; turns on recording for channel C32
3. P1 receives Marker over C21, sets state(C21) = {a}; sends Marker to P2 & P3; turns on recording for channel C23
4. P3 receives Marker over C13, records its state (S3), sets state(C13) = {}; sends Marker to P1 & P2; turns on recording for channel C23
5. P2 receives Marker over C32, sets state(C32) = {b}
6. P3 receives Marker over C23, sets state(C23) = {};
7. P1 receives Marker over C31, sets state(C31) = {};

Diagram of state transitions and message exchanges among processes P1, P2, and P3 during the snapshot process.