Synchronization in Distributed Systems

Cooperation and Coordination in Distributed Systems

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

But... not enough for cooperation:
• Synchronization
  • Deadlock avoidance
  • Consistency in transaction processing
  • Groups of replicated objects

More complicated problems than in central systems!

Kinds of Synchronization

• Synchronization based on actual (absolute) time
• Synchronization 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 synchronized
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 – synchronization is needed considering the absolute time on all machines:

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. $\frac{dC}{dt} = 1$
  \[ \Rightarrow \text{Reality: there is a clock drift so that a maximum drift rate can be specified:} \]
  \[ \rho : 1 - \rho \leq \frac{dC}{dt} \leq 1 + \rho \]

- Needed for synchronization: definition of a tolerable skew, the maximum time drift $\delta$
- With this, re-synchronization has to be made in certain intervals: all $\delta/2\rho$ seconds
- How to make such a re-synchronization?
### Cristian’s Algorithm

- There is one central **time server** $T$ with a UTC receiver
- All other machines $M$ are contacting the time server at least all $\delta_2p$ seconds
- $T$ responds as fast as it can

$M$ computes current time:

- Hold time $t_{send}$ for sending the request
- Measure time when response with $t_{UTC}$ arrives ($t_{receive}$)
- 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

### Berkeley Algorithm

Another approach (Berkeley Unix):

1. **active time server**
2. **logical synchronization**

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 **centralized server**, mainly used in Intranets

Simple mechanism for decentralized synchronization (based on Berkeley Algorithm):

- Divide time into fixed-length synchronization 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: synchronization subnet tree

Primary servers are connected to UTC sources

Secondary servers are synchronized to primary servers (Synchronization subnet)

Lowest level servers in users’ computers, synchronised to secondary servers

Note: this is only an example, there can be more than three layers
NTP - Synchronization of Servers

- The synchronization 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 synchronization:
  - **Multicast**
    - A server within a 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 when very high accuracy is needed (e.g. for higher levels)

- All modes use UDP to transfer time data

Accuracy of NTP

- For each pair of messages between two servers, NTP estimates
  - an offset \( o \) between the two clocks and
  - a delay \( d \) (total time for transmitting the two messages, which take \( t \) and \( t' \)).
  - You have: \( T_{i-2} = T_{i-3} + t + o \) and \( T_i = T_{i-1} + t' - o \)
  - for the current offset \( o \) between A and B

- This gives us (by adding the equations):
  \[ d = t + t' = T_{i-2} - T_{i-3} + T_i - T_{i-1} \]

- Also (by subtracting the equations)
  \[ o = o_i + (t' - t)/2 \] where \( o_i = (T_{i-2} - T_{i-3} - T_i + T_{i-1})/2 \)

Accuracy of NTP

- Using the fact that \( t, t'>0 \) it can be shown that
  \[ o_i - d/2 \leq o \leq o_i + d/2 \]

- Thus \( o \) is an estimation of the offset and \( d \) is a measure of the accuracy

- NTP servers filter pairs \(<o, d>\), estimating reliability of time servers from variations in pairs and accuracy of estimations by low delays \( d \), 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 (Concurrent Processes/Events, notation: \( a \| b \))

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

Application of Lamport Timestamps

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

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 \( P_i \)
   - 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 \( m \)
   - \( P_j \) adjusts its own vector for each \( k \) to \( V[j] = \max(V[j], V_i[j]) \) (These entries reflect the number of messages that \( P_i \) has to receive to have at least seen the same messages that preceded the sending of \( m \))
   - Add 1 to entry \( V[j] \) for the event of receiving \( m \)

Unsynchronized clocks: messages C and D arrive before they are sent

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. 0805200614053022.1300)
### 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 $V_i$ on every message it sends.
- When $p_j$ receives $(m, V)$ it sets $V[j][k] := V[j][k] + 1$ for the receiving event and afterwards $V[k][i] := \max(V[k][i], V[j][k]) \quad k = 1, 2, \ldots, N$.

#### Host 1
- 0,0,0,0
- 0,0,0,0
- 0,0,0,0
- 1,0,0,0
- 2,0,1,0
- 2,0,2,0
- 2,0,2,1
- 2,0,2,2

#### Host 2
- 2,0,0,0
- 2,0,0,0
- 2,0,0,0
- 2,0,2,0
- 2,0,2,0
- 2,0,2,0
- 2,0,2,0
- 2,0,2,0

#### Host 3
- 0,0,0,0
- 2,0,1,0
- 2,0,2,0
- 2,0,2,0
- 2,0,2,0
- 2,0,2,0
- 2,0,2,0
- 2,0,2,0

#### Host 4
- 0,0,0,0
- 2,0,0,0
- 2,0,0,0
- 2,0,0,0
- 2,0,0,0
- 2,0,0,0
- 2,0,0,0
- 2,0,0,0

### 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.

### Detecting Causality Violation

- Potential causality violation can be detected by vector timestamps.
- If the vector timestamp of a message is less than the local vector timestamp, on arrival, there is a potential causality violation.
**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

- **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
  - Both processes are passive and seem to be terminated, but in fact there is a message sent by `p_2` to activate `p_1`

- **c. Termination**
  - `P_1` and `P_2` are passive

**Examples:**

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

**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 initialize 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`

**Consistent cut**

- Allows sent messages

**Inconsistent cut**

- Allows no received-but-not-sent messages
Snapshot Algorithm of Chandy/Lamport

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 $c$).