5. Models of Distributed Computation

5.1. Causality

Distributed systems lack of a global state, their nature is asynchronous.

- Non-instantaneous communication
  - Different observers may observe the same event at different times and different events at the same time
  - Reason: propagation delay, contention for network resources, retransmission (due to lost messages) etc.

- Relativistic effects
  - Synchronizing by time is unreliable
  - Reason: clocks tend to drift apart

- Interruptions
  - Even if two computers receive a message at the same time, their reaction may need different time
  - Reason: Complex computer systems with unpredictable execution times due to CPU contention, interrupts, page faults, cache misses, garbage collection etc.

*Distributed systems are causal: the cause precedes the effect (if traveling backward in time is excluded).*

- Suppose the distributed system is composed of the set of processors $P = \{p_1, \ldots, p_m\}$.

- The set of all events of a distributed system is $E$, the set of all events of processor $p$ is $E_p$. 
5.1.1. Happens Before

- If event $e_1$ occurred before $e_2$, we write: $e_1 < e_2$ (or $e_1 \rightarrow e_2$). We say: $e_1$ happened before $e_2$

- If we know this based on the information $I$, we write: $e_1 <_I e_2$

- The events of the same processor are totally ordered
  - If $e_1 \in Ep$ and $e_2 \in Ep$; either $e_1 <_p e_2$ or $e_2 <_p e_1$

- Sending of a message happens always before receiving it
  - If $e_1$ is the sending of message $m$ and $e_2$ the receipt of $m$, then $e_1 <_m e_2$

- Happened-before relation: The transitive closure of the processor and message passing orderings:
  - If $e_1 <_p e_2$ then $e_1 <_H e_2$
  - If $e_1 <_m e_2$ then $e_1 <_H e_2$
  - If $e_1 <_H e_2$ and $e_2 <_H e_3$ then $e_1 <_H e_3$

- An event $e_1$ happened before $e_2$ if there is a causally linked chain of events leading $e_1$ from to $e_2$. That means, $e_1$ could have caused $e_2$.

- The happened-before relation is a partial order but (usually) not a total order:
  - It is possible to have two events $e_1$ and $e_2$ that neither $e_1 <_H e_2$ nor $e_2 <_H e_1$ we denote: $e_1 \parallel e_2$
  - Such events are called concurrent (or disjoint)
Example

- \( e_1 \prec_{p1} e_4 \prec_{p1} e_7 \) (all events on the same processor)
- \( e_2 \prec_{p2} e_3 \prec_{p2} e_5 \) (all events on the same processor)
- \( e_1 \prec_m e_3 \) (send and receipt if the same message)
- \( e_5 \prec_m e_8 \)
- \( e_1 \prec_H e_3 \prec_H e_5 \prec_H e_8 \), etc. \( \Rightarrow e_1 \prec_H e_8 \) (transitivity)
- \{e_1, e_6\}, \{e_1, e_2\} \{e_2, e_6\}, are concurrent
- Happens-before DAG (H-DAG)
  - The vertices are the events in \( E \)
  - The directed edge \((e_1, e_2)\) is in the edge set, \( E_H \) iff (if and only if) \( e_1 \prec_p e_2 \) or \( e_1 \prec_m e_2 \).
5.1.2. Lamport Timestamps

• A logical global clock assigns a total order over all events
• The happened-before relation defines a partial order
  ➢ Theoretically we can just apply topological sort on $<_H$
• Leslie Lamport’s algorithm (Lamport78)
  ➢ creates total order “on the fly”
  ➢ entirely distributed
  ➢ fault tolerant
  ➢ efficient
  ➢ orders concurrent events arbitrarily

➢ Each event $e$ has a timestamp $e.TS$
➢ Each processor maintains a local timestamp $my.TS$
➢ The processor address (or number) is used for the lowest order bits of the timestamp (to avoid identical stamps in different processors)
➢ Each event and each message get a timestamp assigned as:

\[
my.TS = 0; \quad // \text{initial assignment}
\]

On event $e$,

if $e$ is the receipt of message $m$,
\[
my.TS = \max(m.TS, my.TS)
\]
//local stamp “jumps” forward, if $m$ is in the “future”
my.TS++
e.TS = my.TS
if $e$ is the sending of message $m$,
m.TS = my.TS

➢ From this follows: if $e_1 <_H e_2$ then $e_1.TS < e_2.TS$, because
  ▪ if $e_1 <_p e_2$ or $e_1 <_m e_2$, $e_2$ is assigned a higher timestamp than $e_1$
Example

But: $e_1.TS < e_2.TS$ then $e_1 <_H e_2$ does not hold always
It holds for the example above, but if we introduce a new event on $p_3$ for instance then we may build a counter-example.

Which one? (try to derive a happens before relationship for two events which are in reality concurrent).
5.1.2. Vector Timestamps

- Lamport algorithm guarantees: $e_1 <_H e_2 \Rightarrow e_1.TS < e_2.TS$
- It does not guarantee: $e_1.TS < e_2.TS \Rightarrow e_1 <_H e_2$
  - Because: concurrent events are ordered arbitrarily
  It would however be practical to detect (we can compute timestamps in a distributed system) if 2 events happens before or not (i.e., they are concurrent).

We introduce a timestamp $VT$ with comparison function $<_V$ such that $e_1 <_H e_2$ iff $e_1.VT <_V e_2.VT$
(introduced by Mattern89 and Fidge91)

Features:
- $<_V$ must be a partial order (since $<_H$ is)
- $e.VT$ must contain information about the other processors
- We need a vector of integers of size $N$ (no. of processors)
- If $e.VT[i] = k$ then $e$ causally follows the first $k$ events of processor $i$ (per definition, an event follows itself)
- $e_1.VT \leq_V e_2.VT$ iff $e_2$ follows every event that $e_1$ follows
  - $e_1.VT \leq_V e_2.VT$ iff $e_1.VT[i] \leq e_2.VT[i] \ \forall \ i = 1, \ldots, N$
  - $e_1.VT <_V e_2.VT$ iff $e_1.VT \leq_V e_2.VT$ and $e_1.VT # e_2.VT$

```plaintext
my_VT = [0, ..., 0]; // initial assignment
On event e,
  if e is the receipt of message m,
    for i = 1 to N
      my_VT[i] = max(m.VT[i], my_VT[i])
    // VT[i] "jumps" forward, if m is in the "future"
    my_VT[self]++
  e.VT = my_VT
  if e is the sending of message m,
    m.VT = my_VT
```

Distributed Systems, 5. Causality 6  László Bőszörményi and Harald Kosch
• Proof

\[ e_1 <_H e_2 \Rightarrow e_1.VT <_{VT} e_2.VT, \text{ because the algorithm ensures that for every event } e_1 <_p e_2 \text{ or } e_1 <_m e_2: \]
\[ e_1.VT <_{VT} e_2.VT \text{ (very similar to Lamport’s algorithm)} \]

The opposite direction is proofed by assuming the contradictory.

Example:

\[ e_1.VT = (5, 4, 1, 3) \]
\[ e_2.VT = (3, 6, 4, 2) \]
\[ e_3.VT = (0, 0, 1, 3) \]
\[ e_1 \text{ and } e_2 \text{ are concurrent (no path } e_1 \rightarrow e_2 \text{ or } e_2 \rightarrow e_1 \text{)} \]
  \[ \Rightarrow e_1 \parallel e_2 \]
\[ e_3 <_H e_1 (e_1 \text{ follows } e_3): \]
  \[ e_3.VT <_{VT} e_1.VT \Rightarrow e_3 <_H e_1 \]
We often need the causality relation (e.g. to test violation)

- **Causality violation**: the effect arrives earlier than the cause (potential cause)
- **Example** (object $O$ can migrate between processors)

- The request of $p_3$ causally follows the transfer of $O$ from $p_1$ to $p_2$, but is processed before the transfer at $p_2$
- $s(m_1) <_H s(m_3)$ but $r(m_3) <_{p_2} r(m_1)$ – how to detect?
• Let \( s(m) \) be the sending and \( r(m) \) the receipt of message \( m \)

• \( m_1 \) causally precedes \( m_2 \) (\( m_1 \prec_c m_2 \)) if \( s(m_1) \prec_H s(m_2) \)

• *Causality violation:* \( m_1 \prec_c m_2 \), but \( r(m_2) \prec_p r(m_1) \):
  ➢ Message \( m_1 \) is sent before \( m_2 \), but \( m_2 \) received before \( m_1 \)

To *detect* causality violation, we can use the timestamp \( VT \)
  ➢ Example (object \( O \) can migrate between processors)

\[
(1,0,0) \prec_V (3,2,3) \text{ !}
\]

➢ We do not avoid causality violation, we just detect it
\( s(m_1) \cdot VT \prec_V (3,2,3) \) and we detect all happens before,
especially:
\[ \Rightarrow s(m_1) \prec_H s(m_3) \text{ but } r(m_3) \prec_p r(m_1) \]

➢ To avoid it, we could e.g. buffer all messages that are not
in order
5.1.3. Causal Communication

- The communication subsystem should prevent causality violation
- General solution: reliable causal multicast
  - Example: 3 processes send each other broadcast messages
    - Every user should get the messages in the same order

We increment the timestamps only at sending

p₃ notices that p₁ has already seen a message from p₂, and buffers m₂, until m₁ arrives
5.1.4. Distributed Snapshots

The global state of a distributed system cannot be determined

- Distributed snapshot
  - A global view of the system consistent with causality
  - Useful in many cases, e.g.:

- Distributed deadlock detection
  - Process p₁ acquires resources r₁ and r₂, then releases r₂
  - Process p₂ acquires r₂ and then requests r₁
  - The deadlock detector observes:
    - p₁ after sending the request for r₂, but before getting it
    - r₂ already after getting the release message from p₁
    - p₂ after sending the request for r₁
    - It detects a **phantom deadlock**

*No consistent cut.* **Consistent:** If a processor P receives a message from Q, the event of sending of Q should be in the cut!

![Diagram of distributed deadlock detection](image-url)
• **Example: Lost token detection**

- Processes p and q pass a privilege token between themselves (my_turn ↔ your_turn)
- We test p and discover that p does not own the token
- Before we can probe q, it sends the token to p
- We falsely conclude that the token is lost

![Diagram showing processes p and q and token states](image-url)
• Global system state
  ➢ N processors: $p_1, ..., p_N$
  ➢ At a given point in time, $p_i$ is in state $s_i \in S = (s_1, ..., s_N)$
    ▪ Theoretically the entire memory
    ▪ Practically it consists of a few important variables
  ➢ Communication channel $C_{i,j}$: $p_i \rightarrow p_j$
    ▪ A $C_{i,j}$ is an ordered list of messages $L_{i,j} = (m_1, ..., m_k)$
    ▪ Communication channels are reliable
    ▪ Communication channels deliver the messages in order
    ▪ The state of a channel contains messages and their order
    ▪ The message at head is delivered first, at tail last
  ➢ Collection of all channels: $C = \{C_{i,j} \mid i, j, \in 1 ..., N\}$
  ➢ Collection of their content: $L = \{L_{i,j} \mid i, j, \in 1 ..., N\}$
  ➢ Global state: $G = (S, L)$

• A global state must be one that could have really occurred
  (we have to build a consistent cut)
  ➢ Suppose we observe $p_i$ (in $s_i$) having received $m$ from $p_k$
  ➢ Then we must observe $p_k$ when it already sent $m$,
    otherwise, we see a message that was not sent
  ➢ The proper time for a snapshot is, when we do not “cut through” any causal chain

• Stable Properties
  ➢ Suppose that global state $G$ is a consistent cut
  ➢ A property $P$ is stable if $P(G) \Rightarrow P(G\prime)$, $\forall G <_c G\prime$
  ➢ E.g. deadlock is a stable property – if occurred
Algorithm of Snapshot

- Organization of a process and channels for a distributed snapshot

- Process Q receives a marker for the first time and records its local state
- Q records all incoming message
- O receives a marker for its incoming channel and finishes

A process has finished, if in all incoming channels the marker are received, and then the result is send back to the initiating process.
• **Example: Lost token detection**

- Process \( p \) and \( q \) exchange a privilege token (●)
- There are two possible processor states: \{\} or \{●\}
- The channels are reliable (keep order)

1) \( p \) requests a snapshot (\( t \)) and records state = \{\}
2) \( q \) sends concurrently the privilege token
3) snapshot token arrives at \( q \): it records state = \{\}, \( L_{pq} = \{\} \)
4) \( q \) sends the snapshot token to \( p \)
5) \( p \) receives privilege token first (channels are reliable)
6) \( p \) receives \( t \) and records: \( L_{qp} = \{●\} \)
7) Observed state: the privilege token is on the way \( q \rightarrow p \)
A few pattern observations

**Termination Algorithms with CONTINUE message:**
- The snapshot may show a global state in which messages are still in transit, i.e. the distributed computation is not yet terminated!
- We use CONTINUE messages: The CONTINUE message is sent if Q has not received any message between the point it recorded the state and the point it has received the marker along each of its incoming channels.

**Termination Algorithms with a tree structure:**
- Many distributed algorithms impose a tree structure
- The tree structure can be used to a *termination detection*
  - The root is the process that initiates the snapshot
  - q is the parent of p if p first learns r’s snapshot from q
  - If a process receives a token it has received before, it answers with *done*
  - If a process receives *done* messages from all processes it sent a token to, it sends a *done* message to the first that sent it a token
  - When the root receives N done messages, the snapshot has finished

**Version numbers**
- Often used to distinguish operations of the same name
- E.g. the snapshot tokens of different initiator processors