Models of Distributed Computation

Distributed System (DS) – overview

- A distributed system is a collection of computers that are spatially separated and do not share a common memory
- Processes executing on those computers communicate with one another by exchanging messages (msg) over communication channels of arbitrary transmission delays
- DS has inherent limitations caused by
  - lack of common memory
  - lack of a system-wide common clock

⇒ Learn how to overcome these inherent limitations

Sequential programs have a simple model of computation

- one global state
- one instruction is executed at a time

Model of computation for distributed program is complex due to

- concurrent executing components
- no global time and no global state (inherent limitations)
- possible failure of components

Approaches to modeling a distributed computation
⇒ causality and consistent state
Fundamental property of distributed system ⇒ **lack of a global state**

Why?? – from an observer’s point of view

- **non-instantaneous communication**

  ![Diagram showing non-instantaneous communication between A, B, and C]

  ⇒ observer’s view of the global state of the system depends on the observation point

- **drift of clock**

  ![Diagram showing clock drift]

  ⇒ real clocks tend to drift apart in their readings

  - **interruptions** – CPU contention, page fault, cache miss, *etc.*

  ⇒ no guarantee of *simultaneous reactions* from different processors

**We cannot count on simultaneous observations (by different observers) of global states in a distributed system**

⇒ So, what properties can we depend on???
Causality

Distributed systems are causal

Causality – *the cause precedes the effect*

*e.g.* sending of a msg precedes the receipt of a msg

Event

- sending of a msg
- receipt of a msg
- local action

Notations

- $\varepsilon$ – the set of all events
- A distributed system is composed of $M$ processors: $p_1, p_2, \cdots, p_M$
- $\varepsilon_p$ – all events that occur at processor $p$

Orders between events — $e_1$ occurs before $e_2$ is denoted as $e_1 < e_2$

- Processor Ordering (PO) – events that occur on the same processor are *totally* ordered
  
  if $e_1 \in \varepsilon_p$ and $e_2 \in \varepsilon_p$
  
  $\implies$ either $e_1 <_p e_2$ or $e_2 <_p e_1$

- Message Passing Ordering (MPO)
  
  $[e_1 – \text{sending of msg } m]$ and $[e_2 – \text{receipt of msg } m]$
  
  $\implies e_1 <_m e_2$
Happened-Before relation ($<_H$) – transitive closure of PO and MPO

- 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 $e_1 <_H e_3$

$e_1 <_H e_2$ if $\exists$ a causally linked chain of events leading from $e_1$ to $e_2$

$\Rightarrow$ a partial order

$\Rightarrow$ if two events are not ordered by $<_H$, they are concurrent ($e_1 \parallel e_2$)

$\Rightarrow$ as a Directed Acyclic Graph (DAG)

Global time does not exist in a DS

$\Rightarrow$ need to construct a global clock (a system of local clocks)

$\Rightarrow$ the clock assigns a total order (TO) to events such that the TO imposed by the global clock is consistent with the PO imposed by $<_H$

$\Rightarrow$ fair request arbitration based on who asked first
Lamport’s Logical Clock Algorithm \(\implies\) create the TO on-the-fly

- each \(e\) has a timestamp \((e.TS)\) and each \(p\) maintains a local TS \((my.TS)\)

- algorithm \(\implies\) completely distributed

\[
\begin{align*}
my.TS &= 0; \\
\text{On event } e, & \\
\quad \text{if } (e == \text{receipt of msg } m) \quad my.TS &= \max(m.TS, my.TS); \\
\quad my.TS++; & \\
\quad e.TS &= my.TS; \\
\quad \text{if } (e == \text{sending of msg } m) \quad m.TS &= my.TS;
\end{align*}
\]

- The timestamps are causal, i.e. \(\text{if } e_1 \prec_H e_2 \text{ then } e_1.TS < e_2.TS\)

- However, timestamps alone do not assign a total order

\[\implies\text{ break ties via processor ID } (p) \implies \text{TO with timestamp } (e.TS, p)\]
Distributed Mutual Exclusion

• use Lamport’s TS to ensure fair access to the critical section (CS)
  – record the time when you request entry into the CS
  – ask everyone else if you are allowed to enter the CS
  – a remote P will agree that you should enter the CS if you made your request earlier than P made its request otherwise, P will deny your request
  – if everyone agrees with your request, you can enter the CS

• why does the algorithm work?
  – Lamport’s TS serves as the global clock
    ▷ total order – the requests can be globally ranked
    ▷ causal – a P that hasn’t requested the CS is ranked lower than a P that has (the receipt of your request will advance someone else’s clock and that someone else’s future request will be ranked lower than your current request)

• algorithm –
  – request_CS – send request to all the other P
  – every other P must reply, which carries the TS at which the replier requested the CS or a null value if the replier is not waiting for or using the CS.
  – release_CS – inform every other P once exit the CS

  ⇒ every P construct the same priority queue of processors waiting to enter the CS
  ⇒ when a P finds that it is at the head of the queue, it knows that no other P will think that it is at the head of the queue, so it is safe to enter the CS
Request_CS() // request critical section
{
    my_timestamp = current_timestamp; // current_timestamp == Lamport’s TS
    is_requesting = TRUE;
    reply_pending = M - 1; // M == total # of processes
    for every other process j
        send(j, REMOTE_REQUEST, my_timestamp);
    wait until reply_pending is 0;
}

Release_CS() // release critical section
{
    is_requesting = FALSE;
    for j = 1 through M
        if (reply_deferred[j] == TRUE) {
            send(j, REPLY);
            reply_deferred[j] = FALSE;
        }
}

CS_Monitor() // critical section monitoring thread
{
    wait until a REMOTE_REQUEST or a REPLY msg is received {
        if REMOTE_REQUEST(sender; request_timestamp) is received {
            let j be the sender of the REMOTE_REQUEST msg;
            if (NOT is_requesting OR my_timestamp > request_timestamp)
                send(j, REPLY);
            else
                reply_deferred[j] = TRUE;
        }
        if REPLY(sender) is received
            reply_pending = reply_pending - 1;
    }
}

Compute() // computation thread
{
    while (1) { sleep(RANDOM); Request_CS(); sleep(RANDOM); Release_CS(); }
}