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

- **drift of clock**
  ⇒ 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 preceeds 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

- $\mathcal{E}$ – the set of all events
- A distributed system is composed of $M$ processors: $p_1, p_2, \cdots, p_M$
- $\mathcal{E}_p$ – 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 $e_1 \in \mathcal{E}_p$ and $e_2 \in \mathcal{E}_p$

  $\implies$ 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 $e_1 <_H e_3$

$\Rightarrow e_1 <_H e_2$ 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{verbatim}
my.TS = 0;
On event e,
  if (e == receipt of msg m) my.TS = max(m.TS, my.TS);
  my.TS++;
  e.TS = my.TS;
  if (e == sending of msg m) m.TS = my.TS;
\end{verbatim}

- The timestamps are causal, i.e.  
  if $e_1 <_H e_2$ then $e_1.TS < e_2.TS$
- However, timestamps alone do not assign a total order
  \implies break ties via processor ID ($p$) \implies TO with timestamp ($e.TS, p$)