Instructor: Paul C. Attie
American University of Beirut

Time, clocks and the ordering of events in a distributed system

1 Overview

Seminal paper by Leslie Lamport:

Time, clocks and the ordering of events in a distributed system. Leslie Lamport. CACM 21(7), 1978.

Defined the very important concepts of "causal order" and "logical clock". Very widely used in distributed computing.

Key idea: define a *happened before* relation $\rightarrow$ on events in a distributed system.

Event $a \rightarrow b$ means a "happened before" b $\rightarrow$ is a binary relation on the set of events that occur in a (computation of) a distributed system.

$\rightarrow$ can be "computed" online by useing *logical clocks*.

System model: a fixed number of sequential processes communicating by asynchronous message passing. Each process is a sequence of events. An event can be either a local computation event or a message send or a message receive. We are not interested in analyzing local computation, so we allow a local computation event to be anything from the execution of a single instruction to the execution of a procedure.

2 The "happened before" relation $\rightarrow$

Definition. $a \rightarrow b$ is the smallest irreflexive binary relation that satisfies all of the following:

1. a and b are events in the same process, and a precedes b.
2. a is the send of a message, and b is the receipt of the same message.
3. there exists event c such that $a \rightarrow c$ and $c \rightarrow b$.

Clause 3 implies that $\rightarrow$ is transitive.

Roughly, $a \rightarrow b$ means that a "can affect" b. Thus, $\rightarrow$ is sometimes called "causal dependence".

Definition. If $a \not\rightarrow b$ and $b \not\rightarrow a$, then a and b are *concurrent*.

We assume that $a \not\rightarrow a$, i.e., an event does not precede itself. Thus $\rightarrow$ is an irreflexive partial ordering on the events in a system.

Causal ordering: $a \rightarrow b$ iff it is possible for a to causally affect b, e.g., by producing a value that is used by b. If a and b are concurrent, then neither can affect the other.

2.1 Connection to Special Relativity

Special relativity defines causality in terms of the mesages (signals) that *could* be sent — the light cone idea. Lamports defines it using the messages that actually *are* sent. This makes more sense for a software system.

3 Logical Clocks

Clock C_i for process P_i is a mapping of P_i's events to numbers: event a of P_i mapped to $C_i(a)$.

Global clock C: $C(a) \triangleq C_i(a)$ iff a is an event of P_i, Useful notation when the process is unspecified.

A system of clocks is correct iff it satisfies:

Clock condition: if $a \rightarrow b$ then $C(a) < C(b)$

We ensure the clock condition by ensuring in turn:

C1. If a and b are events in process P_i and a comes before b, then $C_i(a) < C_i(b)$

C2. If a is the sending of a message by process P_i and b is the receipt of that message by process P_j, then $C_i(a) < C_j(b)$.

We implement C1 and C2 using implementations rules IR1 and IR2 respectively, which are given below. We use the variable $clock_i$ to store the "current" clock valueof process P_i.

IR1. Each process P_i increments $clock_i$ between any two successive events.

To meet condition C2, we require that each message m contain a timestamp which equals the time at which the message was sent. Upon receiving a message, a process must advance its clock to be later than the timestamp:

IR2. (a) If event a is the sending of a message m by process P_i, then the message m contains timestamp $C_i(a)$.
(b) Upon receiving a message m, process P_j sets $clock_j$ greater than or equal to its present value and greater than the timestamp of m.

4 Imposing a Total Order on Events

Sometimes having a total order $\Rightarrow$ on events is useful. Do this by ordering events accorind to the logical times, and using a static ordering on process names to break ties:

Definition. Let a be an event in process P_i and b be an event in process P_j. Then $a \Rightarrow b$ if and only if either (i) $C_i(a) < C_j(b)$ or (ii) $C_i(a) < C_j(b)$ and $i < j$.

$\Rightarrow$ extends the partial order $\rightarrow$ into a total order.

4.1 Distributed Mutual Exclusion

Can use to solve distributed mutual exclusion problem: all request and release operations (of the critical reqsource) by a process are broadcast to all processes, which use $\Rightarrow$ to define a total

order. Requests are granted in the order of $\Rightarrow$, with older requests being granted first (this assures starvation-freedom). Actual implementation uses queues. See Lamports paper for details.

The key idea is that P_i gets the resource when it knows that its request is older (i.e., smaller w.r.t. $\Rightarrow$) than any other request that it has received, and it has received a later message from every other process (i.e., follows P_i's request in $\Rightarrow$). This latter condition means that P_i knows that it will not ever receive a request from another process that is earlier than its own outstanding request. Hence P_i knows that its requests is the oldest outstanding request in the system, and so can go ahead and grant the request.