Lecture 4

15th January 2020

Team 6:Amrit Sahai(2019201009),Hetav Panchani(2018101016),Devansh Gupta(20171100)

PRESENTATION SCRIBE

TOPIC:How vector clock deals with out of order messages ?

CHALLENGES:

1.Recieve the messages in the order they were sent.

2. If a message with higher timestamp is recieved before a lower timestamp message, the protocol should detect that there is a message yet to arrive which was sent earlier.

3. When such a case is present, the current recieved message maybe stored in a buffer until the earlier sent message is recieved first.

PROTOCOLS FOR HANDLING OUT OF ORDER MESSAGES

1.Birman-Schiper-Stephenson Protocol
2.Schiper-Eggli-Sandoz Protocol

Birman-Schiper-Stephenson Protocol

The goal of this protocol is to preserve ordering in the sending of messages. For example, if *send*($m1$) ->*send*($m2$), then for all processes that receive both $m1$ and $m2$,*receive*($m1$) ->*receive*($m2$). The basic idea is that $m2$ is not given to the process until $m1$ is given. This means a buffer is needed for pending deliveries. Also, each message has an associated vector that contains information for the recipient to determine if another message preceded it. Also, we shall assume all messages are broadcast. Clocks are updated only when messages are sent.

Notation

- N processes
- P_i process

- C_i vector clock associated with process P_i; jth element is $C_i[j]$ and contains P_i's latest value for the current time in process P_j
- t^m vector timestamp for message *m* (stamped after local clock is incremented)

Protocol

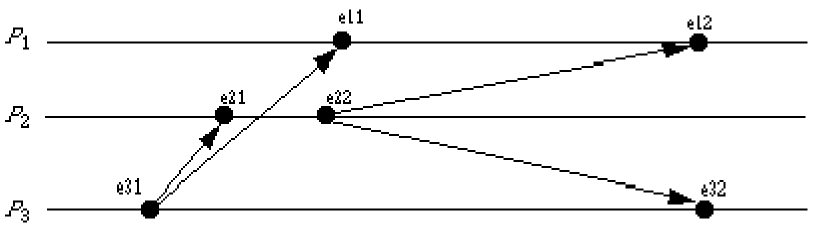
P_i sends a message to P_j

Pi increments *Ci*[*i*] and sets the timestamp t^m =*Ci*[*i*] for message *m*.

P_j receives a message from P_i

- When *Pj*,*j*!=*i*, receives *m* with timestamp t^m, it delays the message's delivery until both
  1. *Cj*[*i*] =t^m[*i*] - 1; and
  2. for all *k*<=*n* and *k* !=*i*,*Cj*[*k*] <=t^m[*k*]
- When the message is delivered to *Pj*, update *Pj*'s vector clock
- Check buffered messages to see if any can be delivered.

EXAMPLE:



Here is the protocol applied to the above situation:

*e*31:P_3sends message *a*; $C_3 = (0, 0, 1)$; $t^a = (0, 0, 1)$

*e*21: P_2 receives message *a*. As $C_2 = (0, 0, 0)$, $C_2[3] = t^a[3] - 1 = 1 - 1 = 0$ and $C_2[1]$ => $t^a[1]$ and $C_2[2]$ => $t^a[2] = 0$. So the message is accepted, and C_2 is set to (0, 0, 1)

*e*11: P_1 receives message *a*. As $C_1 = (0, 0, 0)$, $C_1[3] = t^a[3] - 1 = 1 - 1 = 0$ and $C_1[1]$ => $t^a[1]$ and $C_1[2]$ => $t^a[2] = 0$. So the message is accepted, and C_1 is set to (0, 0, 1)

*e*22: P_2 sends message *b*; $C_2 = (0, 1, 1)$; $t^b = (0, 1, 1)$

*e*12: P_1 receives message *b*. As $C_1 = (0, 0, 1)$, $C_1[2] = t^b[2] - 1 = 1- 1 = 0$ and $C_1[1]$ => $t^b[1]$ and $C_1[3]$ => tb[2] = 0. So the message is accepted, and C_1 is set to (0, 1, 1)

*e*32: P_3 receives message *b*. As $C_3 = (0, 0, 1)$, $C_3[2] = t^b[2] - 1 = 1 - 1 = 1$ and $C_1[1]$ => $t^b[1]$ and $C_1[3]$ => tb[2] = 0. So the message is accepted, and C_3 is set to (0, 1, 1)

Now, suppose t^a arrived as event *e*12, and t^b as event *e*11. Then the progression of time in P_1 goes like this:

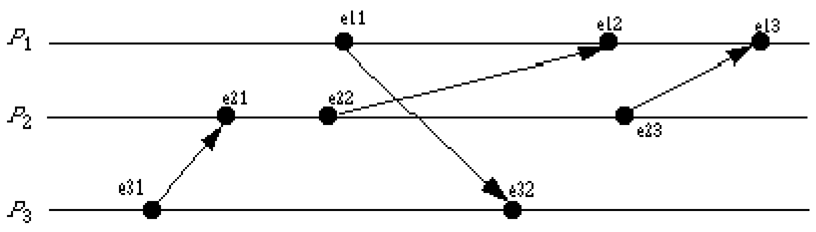
*e*11: P_1 receives message *b*. As C_1= (0, 0, 0), $C_1[2] = t^b[2] - 1 = 1 - 1 = 0$ and $C_1[1]$ =>$t^b[1]$, but $C_1[3] < t^b[3]$, so the message is held until another message arrives. The vector clock updating algorithm is not run.

*E*12: P_1 receives message *a*. As $C_1 = (0, 0, 0)$, $C_1[3] = t^a[3] - 1 = 1 - 1 = 0$, $C_1[1]$ => $t^a[1]$, and $C_1[2]$ => $t^a[2]$. The message is accepted and C_1 is set to (0, 0, 1). Now the queue is checked. As $C_1[2] = t^b[2] - 1 = 1 - 1 = 0$, $C_1[1]$ => $t^b[1]$, and $C_1[3]$ => $t^b[3]$, that message is accepted and C_1 is set to (0, 1, 1).

Schiper-Eggli-Sandoz Protocol

The goal of this protocol is to ensure that messages are given to the receiving processes in order of sending. Unlike the Birman-Schiper-Stephenson protocol, it does not require using broadcast messages. Each message has an associated vector that contains information for the recipient to determine if another message preceded it. Clocks are updated only when messages are sent.

EXAMPLE:



Here is the protocol applied to the above situation:

$e31$: P_3 sends message a to P_2. $C_3 = (0, 0, 1)$; $t^a = (0, 0, 1)$, $V^a = (?, ?, ?)$; $V_3 = (?, (0, 0, 1), ?)$

$e21$: P_2 receives message a from P_1. As $V^a[2]$ is uninitialized, the message is accepted. V_2 is set to $(?, ?, ?)$ and C_2 is set to $(0, 0, 1)$.

$e22$: P_2sends message b to P_1. $C_2 = (0, 1, 1)$; $t^b = (0, 1, 1)$, $V^b = (?, ?, ?)$; $V_2 = ((0, 1, 1), ?, ?)$

$e11$: P_1 sends message c to P_3. $C_1 = (1, 0, 0)$; $t^c = (1, 0, 0)$, $V^c = (?, ?, ?)$; $V_1 = (?, ?, (1, 0, 0))$

$e12$: P_1 receives message b from P_2. As $V^b[1]$ is uninitialized, the message is accepted. V_1 is set to $(?, ?, ?)$ and C_1 is set to $(1, 1, 1)$.

$e32$: P_3 receives message c from P_1. As $V^c[3]$ is uninitialized, the message is accepted. V_3 is set to $(?, ?, ?)$ and C_3 is set to $(1, 0, 1)$.

$e23$: P_2 sends message d to P_1. $C_2 = (0, 2, 1)$; $t^d = (0, 2, 1)$, $V^d = ((0, 1, 1), ?, ?)$; $V_2 = ((0, 2, 1), ?, (0, 0, 1))$
$e13$: P_1 receives message d from P_2. As Vd[1] < $C_1[1]$, so the message is accepted. V_1 is set to $((0, 1, 1), ?, ?)$ and C_1 is set to $(1, 2, 1)$.

Now, suppose t^b arrived as event $e13$, and t^d as event $e12$. Then the progression in P_1 goes like this:

*e*12: P_1 receives message *d* from P_2. But $V^d[1] = (0, 1, 1) <X (1, 0, 0) = C_3$, so the message is queued for later delivery.
*E*13: P_1 receives message *b* from P_2. As $V^b[1]$ is uninitialized, the message is accepted. V_1 is set to (?, ?, ?) and C_1 is set to (1, 1, 1). The message on the queue is now checked. As $V^d[1] = (0, 1, 1) < (1, 1, 1) = C_1$, the message is now accepted. V_1 is set to ((0, 1, 1), ?, ?) and C_1 is set to (1, 2, 1).

LECTURE SCRIBE:

Message ordering:

- FIFO
- NON-FIFO
- CAUSAL

In the FIFO model, each channel acts as a first-in first-out message queue and, thus, message ordering is preserved by a channel. In the non-FIFO model, a channel acts like a set in which the sender process adds messages and the receiver process removes messages from it in a random order. A system that supports causal delivery of messages satisfies the following property: "for any two messages mij and mkj, if

send(mij) → send(mkj), then rec(mij) → rec(mkj)"

Causally ordered delivery of messages implies FIFO message delivery. The causal ordering model is useful in developing distributed algorithms and may simplify the design of algorithms.

Matrix Time

It is the information that a process has about other processes and their information about other processes. In a system of matrix clocks, the time is represented by a set of n×n matrices of non-negative integers. A process pi maintains a matrix mti[1...n, 1...n] where,

- mti[i,i] denotes the local logical clock of pi and tracks the progress of the computation at process pi;

- mti[i,j] denotes the latest knowledge that process pi has about the local logical clock, mtj[j ,j], of process pj (note that row, mti[i..] is nothing but the vector clock vti [.] and exhibits all the properties of vector clocks);
- mti[j,k] represents the knowledge that process pi has about the latest knowledge that pj has about the local logical clock, mtk[k,k], of pk.

The entire matrix mti denotes pi’s local view of the global logical time. The matrix timestamp of an event is the value of the matrix clock of the process when the event is executed. Process pi uses the following rules R1 and R2 to update its clock:

- R1: Before executing an event, process pi updates its local logical time as follows:
  mti[i,i] := mti[i, i] + d (d > 0)
- R2: Each message m is piggybacked with matrix time mt. When pi receives such a message (m,mt) from a process pj, pi executes the following sequence of actions:
  - update its global logical time as follows:
  - $1 \le k \le n$: mti[i, k] := max(mt,[i, k], mt[j, k]), (that is, update its row mti[i, ∗] with pj’s row in the received timestamp, mt);
    - $1 \le k, l \le n$: mti[k, l] := max(mti[k, l], mt[k, l]);
  - execute R1;
  - deliver message m.

Q. Is the principal vector always larger than the non-principal vectors? **TRUE**

This is so because the user has the latest information about its processes. When the information from other processes is received, the principle vector is updated just after receiving and hence, it is always the most updated one.

Global State of a Distributed System

At any instant the state of a process Pi denoted by LSi, is a result of the sequence of all events executed by pi upto that instant.

$SCij = \{mij \mid send(mij) \in LSi \text{ and } rec(mij) \in LSj\}$

To formalize, let LSxi denote the local state of process Pi after the occurrence of all events until event exi

- LS0i is the initial state of Pi

Conditions for a consistent global state

A global state GS = { ⋃iLSxii , ⋃j,kSCyj, zk jk } is a consistent global state iff it satisfies the following condition:

$$\forall mij : send(mij) \ntrianglelefteq LSxii \Rightarrow mij \notin SCxi, yjij \wedge rec(mij) \ntrianglelefteq LSyjj$$

That is, channel state SC yi,zk ik and process state LSzk k must not include any message that process pi sent after executing event eixi

Homework Questions:

Q1. Taking example of using matrix clock (2D) over vector clock (1D) and the significance of the additional information that is sent in matrix clock, do you think adding additional dimension (3D matrix and so on) to the clock will be useful? What more information can be sent by using additional dimensions?

Q2. Consider causal message ordering.

Assume vector clocks being used.
Consider an updation event at site D which was propagated to all sites at time Tsu; which A is aware of.
A wants to know if B got the update by looking at Bs vector sent to A. Can A know?

Now A wants to know if C knows about the update by looking at B’s vector. That is A is interested in Vc[D] but he needs to figure this out looking at B’s vector in the msg from B to A.

Assume that C has sent B a msg recently where VB[C] >= Tsu

Then is it true that C has received the update? Can A figure out if C has received the broadcast?

If the messages passed used matrix clocks instead of vector clocks then can A have answers to both problems above?