


EEC-681/781

Distributed Computing Systems

Lecture 9


Wenbing Zhao
Department of Electrical and Computer Engineering
Cleveland State University
wenbing@ieee.org



Outline

- Announcement:
 - **Midterm #2 (lecture 6-10, lab 3) – Tuesday, 4-6pm, March 28, SH306**
 - Move 04/06's lab session (lab4) to 04/04 6-7:50pm
- Event ordering and logical clocks
 - (Lamport, 1978)


23 March 2006 EEC681/781 Wenbing Zhao



Event Ordering

- “**Time, Clocks, and the Ordering of Events in a Distributed System**”, by Leslie Lamport, Communications of the ACM, July 1978, Volume 21, Number 7, pp.558-565
- He showed that it is possible to synchronize all the clocks to produce a single, unambiguous time standard
- He pointed out the clock synchronization need not to be absolute
 - What usually matters is not that all processes agree on exactly what time it is, but rather, that they agree on the order in which events occur

23 March 2006 EEC681/781 Wenbing Zhao



Happens-Before Relation

- With perfectly accurate physical time
 - An event a happened before an event b if a happened at an earlier time than b
- Without using the physical clocks
 - Assume that the system is composed of a collection of processes, each process consists of a sequence of events
 - The events of a process form a sequence, where a occurs before b in this sequence if a happens before b
 - Assume sending and receiving a message is an event in a process

23 March 2006 EEC681/781 Wenbing Zhao

Happens-before Relation

- “Happens-before” relation, denoted by “ \rightarrow ”, is defined as follows:
 - The relation “ \rightarrow ” on the set of events of a system is the relation satisfying the following three conditions:
 - ◆ If a and b are events in the same process, and a comes before b , then $a \rightarrow b$
 - ◆ If a is the sending of a message by one process and b is the receipt of the same message by another process, then $a \rightarrow b$
 - ◆ If $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$
 - Event a **causally** affects event b

Partial Ordering

- Two distinct events a and b are said to be concurrent if $a \nrightarrow b$ and $b \nrightarrow a$
 - Neither event can causally affect the other
 - This introduces a partial ordering of events in a system with concurrently operating processes

Logical Clocks

- Look the clock just as a way of assigning a number to an event, where the number is the time at which the event occurs
 - Define a clock C_i for each process P_i
 - ◆ Assigns a number $C_i(a)$ to any event a in that process
 - The entire system of clocks is represented by the function C which assigns to any event b the number $C(b)$, where $C(b) = C_j(b)$ if b is an event in process P_j
 - The clocks C_i are logical clocks rather than physical clocks

Implementation of Logical Clocks

- The logical clocks is correct if the events of the system that are related to each other by the *happens-before* relation can be properly ordered using these clocks
- Clock condition:
 - For any event a, b , if $a \rightarrow b$ then $C(a) < C(b)$

Implementation of Logical Clocks

- According to our definition of the *happens-before* relation, the clock condition is satisfied if the following two conditions hold:
 - 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 $C_i(a) < C_j(b)$

Implementation of Logical Clock

- To meet C1:
 - Each process P_i increments C_i between any two successive events
- To meet C2:
 - (a) if event a is the sending of a message m by process P_i , then the message m contains a timestamp $T_m = C_i(a)$.
 - (b) Upon receiving a message m , process P_j sets C_j greater than or equal to its present value and greater than T_m

Implementation of Logical Clocks by Counters

- A **Lamport logical clock** is a monotonically increasing software counter
- Each process P_i keeps its own logical clock C_i which is used to apply **Lamport timestamps** to events
- To capture the **happens-before relation** \rightarrow , processes update their logical clocks and transmit the values of their logical clocks in messages as follows:
 - Before each event at P_i : $C_i := C_i + 1$
 - When P_i sends a message m , it piggybacks $t = C_i$
 - When P_j receives (m, t) : $C_j := \max(C_j, t) + 1$
- $e \rightarrow e' \Rightarrow C(e) < C(e')$

Total Ordering of Events

- We can use the logical clocks satisfying the Clock Condition to place a total ordering on the set of all system events
 - Simply order the events by the times at which occur
 - To break the ties, Lamport proposed the use of any arbitrary total ordering of the processes, i.e. process id

Total Ordering of Events

- Using this method, we can assign a unique timestamp to each event in a distributed system to provide a total ordering of all events
- Very useful in distributed system
 - Solving the mutual exclusion problem
 - Totally ordered reliable multicast => needed to build fault tolerant systems

Vector Timestamps

- Lamport timestamps do not guarantee that if $C(a) < C(b)$ then a indeed happened before b
- We need **vector timestamps** for that
 - Each process P_i has an array $V_i[1..n]$, where $V_i[j]$ denotes the number of events that process P_i knows have taken place at process P_j
 - When P_i sends a message m , it adds 1 to $V_i[i]$, and sends V_i along with m as **vector timestamp** $vt(m)$
=> upon arrival, each other process knows P_i 's timestamp
- $V_i(a) < V_j(b) \Leftrightarrow$
 - $V_i(a)[k] \leq V_j(b)[k]$ for every k , and
 - $V_i(a)[i] < V_j(b)[j]$

Vector Timestamps

- When a process P_j receives a message m from P_i with vector timestamp $vt(m)$, it
 - (1) updates each $V_j[k]$ to $\max(V_j[k], vt(m)[k])$, and
 - (2) increments $V_j[j]$ by 1. **Book is wrong!** (not increment $V_j[i]$ by 1)