Lecture Notes 2
Basic Concepts in
Distributed Computing

Anand Tripathi
CSci 8980
Highly Available and Scalable Distributed Systems

Characteristics of Distributed Systems
• No shared memory
  – Message passing is the sole mechanism for inter-
    process communication
• No global clock
  – Events cannot be scheduled or synchronized
    using a global clock
• Asynchronous vs. Synchronous Model
  – In asynchronous system message communication
    delays and the time required in executing a
    processing step may be unbounded.
  – In synchronous systems, time bounds can be
    specified for these two.

Causality and Logical Clocks
• The notion of logical clocks was introduced by
  Lamport in 1978.
• It allows us to reason about the ordering of events in
  a distributed system in the absence of a globally
  synchronized clock.
• It is based on “happened before relationship”: Two events, a and b, are causally related if either:
  – a happened-before b: a \rightarrow b
  – Or b happened-before a: b \rightarrow a

Logical Clocks (Lamport Clocks)
• With each event a in a distributed system and
  integer clock value is associated. It is given by:
  \[ C(a) \] clock value of event a
• The clock value is also called the timestamp
  of the event.
• The logical clock has the following property:
  \[ \text{If } a \rightarrow b, \text{ then } C(a) < C(b) \]
Implementation of Lamport Clocks

- Each process $P_i$ maintains an integer counter $C_i$.
- For an event $a$ occurring in process $P_i$: $C(a)$ is given by the value of $C_i$ when event $a$ occurs. It is denoted by $C_i(a)$

Implementation Rules:

IR1: Clock $C_i$ is incremented between any two successive events in $P_i$:

$C_i := C_i + d$, where $d$ is an integer and $d > 0$

IR2: When process $P_i$ sends message $m$ to $P_j$, it assigns timestamp $t_m = C_i(a)$ to the message. $P_j$ advances its clock to: $C_j := \max(C_j, t_m) + d$, where $d$ is an integer $> 0$.

A Limitation of Lamport Clock

- If $C(a) < C(b)$, that does not mean that $a \rightarrow b$

Implementation of Vector Clocks

- Each process $P_i$ maintains a vector $C_i$ of $n$ integers.
- Vector clocks do not have this limitation.
- In a system of $n$ processes, the clock value is a vector of $n$ integers:
  
  $VT = \text{Vector clock}$
  
  $VT[i]$ - represents the logical clock value w.r.t. process $P_i$

Comparison of Vector Clocks:

- $T_a = T_b$, iff for all $i$, $T_a[i] = T_b[i]$
- $T_a \neq T_b$, iff for some $i$, $T_a[i] \neq T_b[i]$
- $T_a \leq T_b$, iff for all $i$, $T_a[i] \leq T_b[i]$
- $T_a < T_b$, iff ($T_a \leq T_b$) and ($T_a \neq T_b$)
- $T_a \parallel T_b$, iff ($T_a \leq T_b$) and ($T_a \neq T_b$)

Example

- For $P_1$: $(1,0,0)$
- For $P_2$: $(0,1,0)$
- For $P_3$: $(0,0,1)$

Example

- For $P_1$: $(1,0,0)$
- For $P_2$: $(2,2,0)$
- For $P_3$: $(0,0,1)$
Vector Clock and Causal Relationship

Theorem: Let VT_a and VT_b be the vector clock values for two events a and b.
VT_a < VT_b if, and only if, a  b

Proof:
Part I: a  b implies VT_a < VT_b

a  b means that there exists a sequence e1...en (possibly empty) such that
a  e1  e2  ...  en  b
We can show that the clock values of any successive events ei  ei+1 must have increasing clock values.

Part II: VT_a < VT_b implies a  b

Case 1: a and b occur in the same process i.
This implies that VT_i[i] ≠ VT_i[i]
⇒ VT_i[i] < VT_i[i]
⇒ a  b

Case 2: a occurs in Pi and b occurs in Pj
⇒ VT_i[i] ≤ VT_i[i]
Let m denote the value of VT_i[i].
Process Pj can have its C_i[i] equal to m, or greater than m, only by receiving a message that causally follows a.

Causal Message Communication

- Consider a system of n communicating processes.
- Send_k(m) represents the event of sending message m to process P_k

Causal Delivery of Messages:
Send_k(m1)  Send_k(m2) implies that delivery_k(m1)  delivery_k(m2)

Utility of Causal Message Delivery

- Consider a replicated bank account database.
- All three copies must be kept consistent in the presence of concurrent operations:

Example of Non-causal Delivery

![Graph showing non-causal delivery]
Utility of Causal Message Delivery

Vector Clocks and Causal Delivery

Let \( V_T \) and \( V_T \) be denote the vector clocks at \( P_i \) and \( P_j \) for the message communication shown below.

- Message \( m \) has vector time \( V_T = C_i \)
- \( C_j \) is the vector time when \( P_j \) receives \( m \).

**Part I:** Causal order violation implies \( V_T < C_j \)

**Part II:** \( V_T < C_j \) \( \Rightarrow \) causal delivery order is violated

Applications of Vector Clocks

- Causal communication
  - Broadcast protocol
  - One-to-one communication
- Message stability detection
- Detection of event pattern for debugging
Causal Communication Protocols

- Birman-Schiper-Stephenson Protocol
  - (See reference paper in ACM Trans. on Computer Systems)
  - It is for broadcast based communication in a group of processes

- Schiper-Eglli-Sandoz Protocol
  - For arbitrary point-to-point communication between processes

Birman-Schiper-Stephenson Protocol

- Each process maintain a vector of counters
- It can be viewed as a vector clock, but there are some differences
- \( V_T[i] \) represents the serial number of the last message broadcast by process \( P_i \)
- Each message is assigned a timestamp (sequence number); the numbers are consecutively increasing without any gap.
- \( V_T[j] \) represents the sequence number of the last message received by \( P_i \) from \( P_j \).
- Each message \( m \) is broadcast with the following data:
  - \( (m, V_T[i]) \)

Actions of process \( P_i \) when it broadcasts a message:
1. Before broadcasting a message: \( V_T[i]++ \)
2. \( V_T[i] \) is used as the timestamp \( T_m \) of the message.
3. Broadcast message \( (m, T_m) \)

Actions of process \( P_j \) when it receives a message \( m \) from \( P_i \):
1. Deliver message \( m \) only when the following two conditions are satisfied:
   - \( V_T[j] = T_m - 1 \)
   - For all \( k \neq i \), \( V_T[k] \geq T_m[k] \)
2. Update vector clock of \( P_j \)
   - For all \( k \), \( V_T[k] := \max(V_T[k], T_m[k]) \)

Schiper-Eglli-Sandoz Protocol

- It support causal delivery of messages for peer-to-peer communication in a group of processes.
- Each process \( P_i \) maintains a vector \( C_i \) of up to \((n-1)\) elements of the form \((P_j, T_j)\), at most one element for each of the other \((n-1)\) processes.
- An element \((P_j, T_j)\) in \( C_i \) indicates that:
  - A message with timestamp \( T_j \) has been sent to \( P_j \)
  - the current state of process \( P_i \) causally follows the send operation of the message with timestamp \( T_j \)
  - A message sent by \( P_i \) to \( P_j \) in this state or later should be delivered only after the message with timestamp \( T_j \)
- Each process \( P_i \) maintains a vector clock \( V_T \), which is updated using the normal rules for vector clocks.
Schiper-Eggli-Sandoz Protocol... 3

Actions taken by process \( P_i \) when sending message \( m \) to process \( P_j \):

1. **Send** \((m, T_m, V_m)\) to \( P_j \) such that:
   - \( T_m := V_T_i \)
   - \( V_m := C_i \)

2. **Update** \( C_i \): Insert in \( C_i \) element \((P_j, T_m)\). Any other previously existing element for \( P_j \) in \( C_i \) overwritten.

Schiper-Eggli-Sandoz Protocol... 4

Actions taken by process \( P_j \) when it receives \((m, T_m, V_m)\) from \( P_i \):

- **Case 1:** \( V_m \) does not contain any element for \( P_j \)
  - In this case deliver message \( m \).

- **Case 2:** \( V_m \) contains an element \((P_j, T)\)
  - **Case 2(a):** \( T > V_T_j \)
    - Message cannot be delivered at this point. It must be buffered for later deliver when the message with timestamp \( T \) has been delivered.
  - **Case 2(b):** \( T < V_T_j \)
    - Message can be delivered at this point.

Schiper-Eggli-Sandoz Protocol... 5

Message delivery by \( P_j \) is accompanied with the execution of the following three actions:

1. **Updating of** \( C_j \)
2. **Updating of** \( V_T_j \)
3. **Checking delivery conditions** for any buffered messages.

Schiper-Eggli-Sandoz Protocol... 6

1. **Updating of** \( C_j \):
   - It involves merging of the entries for each process \( P_k \)
     in \( V_m \) and \( C_j \) using the following rules:
     - **Case 1:** An entry \((P_k, T)\) exists in \( V_m \) but there is no entry for \( P_k \) in \( C_j \)
       - In this case, insert \((P_k, T)\) in \( C_j \).
     - **Case 2:** An entry \((P_k, T)\) exists in \( V_m \) and an entry \((P_k, T')\) exists in \( C_j \)
       - In this case insert \((P_k, T'')\) in \( C_j \) such that for all \( i \):
         - \( T''[i] := \max(T[i], T'[i]) \)
   2. **Update** \( V_T_j \) according to the vector clock update rules.

Baldoni and Raynal Paper

- It presents some applications of vector clocks.
  - Message stability
- It also presents drawbacks and limitations of vector clocks:
  - It does not scale — for a large number \( n \) of processes the vector size is large.
  - Certain applications require vector of VTs to be communicated.
- The paper also presents:
  - Approximate vector clock schemes when \( n \) is large and we do not want to have vector size \( n \).

Application of Vector Clocks: Message Stability Detection Problem

- Refer to the article by Baldoni and Raynal in IEEE DS Online.
- Consider the problem of reliable message broadcast in a group.
- A process may fail in between broadcasting a message, or the network may get partitioned during a broadcast.
- We want to ensure that eventually each process delivers a broadcast message.
Application of Vector Clocks
Message Stability Detection

To solve this problem:

• Each process must buffer a copy of every message it sends or receives.
• If a process $P_i$ fails, any process with a copy of a message $m$ sent by $P_i$ can forward $m$ to any process $P_j$ that detects it has not received $m$.

Problem:
When should a process delete a message from its buffer?

• When a message that has been delivered to all its intended destinations, it is not necessary to keep in the buffer.
• Such a message is called a stable message, and we can safely discard such messages from a process's local buffer.

Message Stability Detection

• We will assume that the communication channels are FIFO and no loss of messages in communication.
• We will enforce causal delivery order by using the scheme of Birman-Schiper-Stephenson protocol.
• Each process $P_i$ will maintain a vector clock $V T_i$ such that $V T_i[j]$ will indicate the sequence number of the last message received.
  – All message preceding it from $P_j$ have been
  – Each message carries with it the sender’s vector clock.
  – On receiving a message, the receiver deposits the message in its local buffer and then delivers after applying the causality checks.

Interpretation of Vector Clock Values

![Diagram](image1)

Consider the vector VTs

![Diagram](image2)

Approximate Vector Clocks

• Vector size is $k$, such that $k < n$.
• $V T_a < V T_b$ does not necessarily imply $a \rightarrow b$
• Such implication may be assumed to hold with some probability $< 1$.

Example:
$f_k(i)$ maps a given $i$ in $(1..N)$ to an integer in the range $(1..k)$.
Suppose that $n=4$, and $k=2$, and $f$ is defined such that
$f(1) = f(2) = 1$
and $f(3) = f(4) = 2$

See the paper by Baldoni and Raynal for more details:
• Each process maintains the matrix such as the one shown earlier.
• For this, each process $P_i$ attaches its value of $V T_i$ with the message.
• On receiving a message, the receiver process:
  – updates the matrix row corresponding to the sender process;
  – stores the message in a local buffer
• A message in the buffer can be delivered to the application process according to the conditions described by Birman-Schiper-Stephenson protocol.
• Please note that many details are missing in Baldoni-Raynal description such as when and how a process would get messages that it has not received due to a process crash.
Approximate Vector Clocks

Causal Communication Protocols

Discussions and comments on two papers

- Fundamentals of Distributed Computing: A Practical Tour of Vector Clocks
  - Robert Baldoni, and Michel Raynal

- Securing Causal Relationships in Distributed Systems
  - Michael Reiter and Li Gong

Event Pattern Detection

- Consider the following problem in distributed debugging:
  - Events are marked as either white or black.
  - White events represent application level events such as communication between processes.
  - Black events represent a local state when a local predicate is satisfied.
- Problem: Given two black events $s$ and $t$, determine if there exists another black event in the system satisfying the following predicate:
  $(s \rightarrow u)$ and $(u \rightarrow t)$

Problem with simple vector clocks

We want to detect the following predicate:

$P(s, t) \iff (\exists u: \text{for } s \text{ and } t, P_1(s, u, t) \text{ and } P_2(s, u, t))$

$P_1(s, u, t) \iff (\text{black}(s) \text{ and } \text{black}(u) \text{ and } \text{black}(t))$

$P_2(s, u, t) \iff (s \rightarrow u \text{ and } u \rightarrow t)$

- We will only count "black" events in the vector timestamps, as only the black events are relevant to predicate detection.
- Each process will maintain a vector clock which is incremented only when black events occur.

Pattern Detection Problem

We want to detect the following predicate:

$P(s, t) \iff (\exists u: \text{for } s \text{ and } t, P_1(s, u, t) \text{ and } P_2(s, u, t))$

$P_1(s, u, t) \iff (\text{black}(s) \text{ and } \text{black}(u) \text{ and } \text{black}(t))$

$P_2(s, u, t) \iff (s \rightarrow u \text{ and } u \rightarrow t)$

- We will only count "black" events in the vector timestamps, as only the black events are relevant to predicate detection.
- Each process will maintain a vector clock which is incremented only when black events occur.
Pattern Detection Problem

With each event \( e \) we associate two kinds of vector timestamps.

- Vector timestamp \( VC \) counting only black events.
- An array \( MC[1...n] \) of vector timestamps, such that \( MC[j] \) contains the vector timestamp of the last black event \( P_j \) that causally precedes event \( e \).

Problem with simple vector clocks

\( P(s, t1) \) is false, but \( P(s, t2) \) is true.


Significance of MC array

\( P(s, t1) \) is false, but \( P(s, t2) \) is true.


Rules for maintaining clocks

S1: When process \( P_i \) produces an event \( e \):
- \( VC[i] = VC[i] + 1; \)
- \( e.VC = VC_i \) and \( e.MC = MC_i \)
- \( MC_i[i] = VC_i \)

S2: When process \( P_i \) executes a send event (send \( m \)):
- \( m.VC = VC_i \) and \( e.MC = MC_i \)
- send \( m \) to process \( P_j \)

S3: When a process \( P_i \) executes receive event (receive \( m \)):
- \( VC_i = max ( VC_i, m.VC ) \)
- \( \forall k, MC[k] = max ( MC[k], m.MC[k] ) \)

Rule for Pattern Detection

\( P(s, t) \equiv ( \exists u: for s and t, P_1(s, u, t) and P_2(s, u, t) ) \)

\( P_1(s, u, t) = ( \black(s) and \black(u) and \black(t) ) \)

\( P_2(s, u, t) = ( s \rightarrow u and u \rightarrow t ) \)

We only need to focus on \( P_2(s, u, t) \).

\( P_2(s, u, t) = ( s \rightarrow u and u \rightarrow t ) = ( \exists u: s.VC < u.VC < t.VC ) \)

If such an event \( u \) exists, then it is generated by some process \( P_k \) and belongs to the causal past of \( t \).

\( P_2(s, u, t) = ( \exists k: s.VC < t.MC[k] < t.VC ) \)

Rule for Pattern Detection

\( P_2(s, u, t) = ( \exists k: s.VC < t.MC[k] < t.VC ) \)

We know that \( ( \forall k: t.MC[k] < t.VC ) \).
Therefore, the test for \( P_2(s, u, t) \) reduces to:

\( P_2(s, u, t) = ( \exists k: s.VC < t.MC[k] ) \)