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 \( b \rightarrow a \)
Causal Ordering of Events in Distributed Systems

\( a \rightarrow b \) is defined as:

- If \( a \) and \( b \) are events in the same process, and \( a \) occurs before \( b \).
- If \( a \) is either the event of sending a message \( m \) by some process and \( b \) is the event of receiving the same message by some other process.
Causal Ordering

• **Happened-before** relation is transitive

  • $a \rightarrow b$ and $b \rightarrow c$,  $\Rightarrow a \rightarrow c$

• **Concurrent Events:**
  
  – Two events $a$ and $b$ are concurrent, i.e. $a || b$, if:
    
    • not ($a \rightarrow b$ or $b \rightarrow a$)
    
    • This means that two concurrent events are not related causally.
Logical Clocks (Lamport Clocks)

• With each event a in a distributed system and integer clock value is associated. It is given by:
  \[ C(a) \quad \text{clock value of event a} \]

• The clock value is also called the timestamp of the event.

• The logical clock has the following property:
  If \( a \rightarrow b \), 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 $t_m = C_i$ 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 $a \rightarrow b$
Vector Clocks

- Vector clocks do not have this limitation.
- In a system of n processes, the clock value is a vector of n integers:
  
  \[ V_T \] - Vector clock
  
  \[ V_T[i] \] - represents the logical clock value w.r.t. process Pi

- 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 \((T_b < T_a) \) and \((T_a < T_b) \)
Implementation of Vector Clocks

- Each process $P_i$ maintains a vector $C_i$ of $n$ integers.

**IR1: (Implementation Rule 1):**
Each process $P_i$ increments its vector clock at the occurrences of successive events:

$$C_i[i] := C_i[i] + d, \text{ where } d > 0$$

**IR2: (Implementation Rule 2):**
If $a$ is the event of sending a message from $P_i$ to $P_j$, and $b$ is the event of message reception by $P_j$, then clock $C_j$ of $P_j$ is incremented based on the timestamp $T_m$ of the message by the following rule:

- $P_i$ attaches timestamp $T_m = C_i(a)$
- For all $k$, $C_j[k] := \max(C_j[k], T_m[k])$
Example

(1,0,0)

(0,1,0)

(0,0,1)

P1

P2

P3

e_{11}
e_{12}
e_{13}
e_{21}
e_{22}
e_{23}
e_{24}
e_{31}
e_{32}
Example

P1
(1,0,0) (2,0,0) (3,4,1)

\[ e_{11} \] \[ e_{12} \] \[ e_{13} \]

P2
(0,1,0) (2,2,0) (2,3,1) (2,4,1)

\[ e_{21} \] \[ e_{22} \] \[ e_{23} \] \[ e_{24} \]

P3
(0,0,1) (0,0,2)

\[ e_{31} \] \[ e_{32} \]
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, $a \rightarrow b$

Proof:

Part I: $a \rightarrow b$ implies $VT_a < VT_b$

$a \rightarrow b$ means that there exists a sequence $e_1 \ldots e_n$ (possibly empty) such that

\[
\begin{align*}
a &\rightarrow e_1 \rightarrow e_2 \rightarrow \ldots \rightarrow e_n \rightarrow b
\end{align*}
\]

We can show that the clock values of any successive events $e_i \rightarrow e_{i+1}$ must have increasing clock values.
Vector Clock and Causal Relationship

Part II: $\text{VT}_a < \text{VT}_b$ implies $a \rightarrow b$

Case 1: a and b occur in the same process i.
This implies that $\text{VT}_a[i] \neq \text{VT}_b[i]$

=> $\text{VT}_a[i] < \text{VT}_b[i]$

=> $a \rightarrow b$

Case 2: a occurs in Pi and b occurs in Pj

=> $\text{VT}_a[i] \leq \text{VT}_b[i]$

Let $m$ denote the value of $\text{VT}_a[i]$.

Process $P_j$ can have its $C_j[i]$ 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.
- $\text{Send}_{i,k}(m)$ represents the event of sending message $m$ to process $P_k$.

![Diagram showing communication between Node i and Node k through the network.](image-url)
Causal Message Communication

• Consider a system of \( n \) communicating processes.

• \( \text{Send}_{i,k}(m) \) represents the event of sending message \( m \) to process \( P_k \)

Causal Delivery of Messages:

\[ \text{Send}_{i,k}(m_1) \rightarrow \text{Send}_{j,k}(m_2) \] implies that
\[ \text{delivery}_k(m_1) \rightarrow \text{delivery}_k(m_2) \]
Example of Non-causal Delivery

\[ m_1 \]
\[ m_2 \]
\[ m_3 \]
Utility of Causal Message Delivery

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

<table>
<thead>
<tr>
<th></th>
<th>Balance</th>
<th></th>
<th>Balance</th>
<th></th>
<th>Balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>$100</td>
<td>---</td>
<td>$100</td>
<td>---</td>
<td>$100</td>
</tr>
<tr>
<td>P2</td>
<td></td>
<td></td>
<td>Deposit $100</td>
<td></td>
<td>Add 10% interest</td>
</tr>
<tr>
<td>P3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Utility of Causal Message Delivery

P1
- Add $100
  - Balance $200

P2
- Add $100
  - Balance $220

P3
- Add 10% Interest
  - Balance $210
Vector Clocks and Causal Delivery

Let $VT_i$ and $VT_j$ be denote the vector clocks at $P_i$ and $P_j$ for the message communication shown below.

- Message $m$ has vector time $VT_m = C_i$
- $C_j$ is the vector time when $P_j$ receives $m$.

$$VT_m < C_j \text{ if, and only if, causal delivery order is violated}$$

We cannot make such a statement for Lamport Clocks.
We cannot make such a statement for Lamport Clocks.

Example of Non-causal Delivery

Local Lamport clock > timestamp and there is no violation of causal delivery

Local Lamport clock > timestamp and causal delivery is violated
Prove that: $V T_m < C_j$ if, and only if, causal delivery order is violated.

**Part I: Causal order violation implies $V T_m < C_j$**

At this point $C_j$ is set greater than $V T_m$. 

$m$, $V T_m$
Part II: VTm < Cj => causal delivery order is violated

- Process i is the sender of message m
- LHS => VTm[i] ≤ Cj[i] at the time when m is delivered to Pj

- This means that some other message m' must have been delivered to Pj before the delivery of m and VT_{m'}[i] ≥ VT_m[i]

⇒ Send (m) → Send (m')
⇒ But delivery(m') → delivery(m)
Hence, causal order violation.
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-Eggli-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
- $\text{VT}_i[i]$ 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.
- $\text{VT}_i[j]$ represents the sequence number of the last message received by $P_i$ from $P_j$.
- Each message $m$ is broadcast with the following data:
  - $(m, \text{VT}_i)$
Birman-Schiper-Stephenson Protocol...2

Actions of process $P_i$ when it broadcasts a message:
1. Before broadcasting a message: $VT_i[i]++$
2. $VT_i$ 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:
   1. $VT_j[i] = T_m[i] - 1$
   2. For all $k, k \neq i$, $VT_j[k] \geq T_m[k]$
2. Update vector clock of $P_j$
   For all $k$, $VT_j[k] := \max(VT_j[k], T_m[k])$
Birman-Schiper-Stephenson Protocol...3

Delay delivery
Of this message
Schiper-Eggli-Sandoz Protocol

• It support causal delivery of messages for peer-to-peer communication in a group of n 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 $VT_i$, which is updated using the normal rules for vector clocks.
Schiper-Eggli-Sandoz Protocol...2

$P_i$ $C_i = \{ \}$ $C_i = \{ (P_j, Tm) \}$

Send $m$ to $P_j$
$Tm = VT_i$
$V_m = C_i = \{ \}$

Send $m'$ to $P_k$
$Tm' = VT_i$
$V_m' = C_i = \{ (P_j, Tm) \}$

$P_j$

Send $m''$ to $P_j$
$Tm'' = VT_k$
$V_m'' = C_k = \{ (P_j, Tm) \}$

$P_k$

$C_k = \{ \}$ $C_k = \{ (P_j, Tm) \}$
Actions taken by process Pi when sending message m to process Pj:

1. Send (m, Tm, Vm) to Pj such that:
   - Tm := VT_i
   - Vm := C_i

2. Update C_i: Insert in C_i element (P_j, Tm). Any other previously existing element for P_j in C_i overwritten.
Schiper-Eggli-Sandoz Protocol... 4

Actions taken by process Pj when it receives (m, Tm, Vm) from Pi:

- **Case 1:** Vm does not contain any element for Pj
  - In this case deliver message m.

- **Case 2:** Vm contains an element (Pj, T)
  - **Case 2(a):** T > VTj
    - 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 < VTj
    - Message can be delivered at this point.
Message delivery by $P_j$ is accompanied with the execution of the following three actions:

1. Updating of $C_j$

2. Updating of $VT_j$

3. Checking delivery conditions for any buffered messages.
1. **Updating of $C_j$**:

   It involves merging of the entries for each process $P_k$ in $Vm$ and $C_j$ using the following rules:

   - **Case 1**: An entry $(P_k, T)$ exists in $Vm$ 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 $Vm$ 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 $VT_j$ 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 $Pi$ fails, any process with a copy of a message $m$ sent by $Pi$ can forward $m$ to any process $Pj$ 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 be maintain a vector clock $VT_i$, such that $VT_i[j]$ 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

Message broadcast operation

VT1 = < 3, 4, 2, 2>
VT2 = < 3, 5, 2, 4>
VT3 = < 1, 3, 3, 4>
VT4 = < 2, 2, 3, 4>
Consider the vector VTs

Sequence number of messages from Process 3 That have been seen by other processes

The smallest value in this column means that all processes have seen message numbered 2 and all other preceding messages from Process 3. Therefore there is no need for a process to keep this message in its local Buffer.
• 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 \( VT_i \) with the message.
• On receiving a message, the receiver process:
  – updates the matrix row corresponding to the sender process;
  – Store the message in a local buffer
• A message in the buffer cab 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 a when and how a process would get messages that it has not received due to a process crash.
Approximate Vector Clocks

- Vector size is $k$, such that $k < n$.
- $\text{VT}_a < \text{VT}_b$ 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 $f(3) = f(4) = 2$
Approximate Vector Clocks

P1: <1, 0> <2, 0> <3, 0>
P2: <1, 0> <2, 1>
P3: <0, 1> <0, 2>
P4: <0, 1> <3, 1>
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:
  \[(s \rightarrow u) \text{ and } (u \rightarrow t)\]
Problem with simple vector clocks

Case 1

Case 2:
Solving this problem using array of vector clocks

We want to detect the following predicate:

\[ P(s, t) \equiv ( \exists u: \text{for } s \text{ and } t, P_1(s, u, t) \text{ and } P_2(s, u, t) ) \]

\[ P_1(s, u, t) \equiv ( \text{black}(s) \text{ and black}(u) \text{ and black}(t) ) \]
\[ P_2(s, u, t) \equiv ( s \rightarrow u \text{ and } u \rightarrow t) \]

- 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:

\[ P(s, t) \equiv ( \exists u: \text{for } s \text{ and } t, \ P_1(s, u, t) \text{ and } P_2(s, u, t) ) \]

\[ P_1(s, u, t) \equiv ( \text{black}(s) \text{ and black}(u) \text{ and black}(t) ) \]

\[ P_2(s, u, t) \equiv ( s \rightarrow u \text{ and } u \rightarrow t) \]

- 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 back 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 $P_i$ produces an event $e$:

$$VC_i[i] = VC_i[i] + 1;$$
$$e.VC = VC_i \text{ and } e.MC = MC_i$$
$$MC_i[i] = VC_i$$

S2: When process $P_i$ executes a send event (send $m$):

$$m.VC = VC_i \text{ and } e.MC = MC_i$$
$$\text{send } m \text{ to process } P_j$$

S3: When a process $P_i$ executes receive event (receive $m$):

$$VC_i = \max ( VC_i, m.VC )$$
$$\forall k, MC_i[k] = \max ( MC_i[k], m.MC[k] )$$
Rule for Pattern Detection

\[ P(s, t) \equiv ( \exists u: \text{for } s \text{ and } t, P_1(s, u, t) \text{ and } P_2(s, u, t) ) \]
\[ P_1(s, u, t) \equiv ( \text{black}(s) \text{ and black}(u) \text{ and black}(t) ) \]
\[ P_2(s, u, t) \equiv ( s \rightarrow u \text{ and } u \rightarrow t) \]

We only need to focus on \( P_2(s, u, t) \).
\[ P_2(s, u, t) \equiv ( s \rightarrow u \text{ and } u \rightarrow t) \]
\[ \equiv ( \exists u: s.VC < u.VC < t.VC) \]

If such an event \( u \) exists, then it is generated by some process \( P_k \) and belongs to the causal past of \( t \).
\[ P_2(s, u, t) \equiv ( \exists k: s.VC < t.MC[k] < t.VC) \]
Rule for Pattern Detection

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

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

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