Distributed Operating Systems: Theoretical Foundations

Neeraj Mittal

Department of Computer Science
The University of Texas at Dallas

neerajm@utdallas.edu
Two Important Characteristics

- Absence of Global Clock
  - there is no common notion of time
Two Important Characteristics

- Absence of Global Clock
  - there is no common notion of time

- Absence of Shared Memory
  - no process has up-to-date knowledge about the system
Absence of Global Clock

- Different processes may have different notions of time
Absence of Global Clock

- Different processes may have different notions of time

Earth

Mars

I invented AIDS cure!

I invented AIDS cure!
Absence of Global Clock

- Different processes may have **different notions of time**
  
  ![Diagram showing Earth and Mars with patent officer and I invented AIDS cure message]
  
  **Earth**
  
  **Mars**
  
  **Patent Officer**
  
  **Who did it first?**
Absence of Global Clock

- Different processes may have different notions of time

Problem: How do we order events on different processes?
Absence of Shared Memory

- A process does not know current state of other processes
Absence of Shared Memory

- A process does not know **current state of other processes**

```
Bob  Harry
What is Harry doing at present?
```

Problem: How do we obtain a coherent view of the system?
Absence of Shared Memory

- A process does not know **current state of other processes**
Absence of Shared Memory

- A process does not know **current state of other processes**
Absence of Shared Memory

- A process does not know **current state of other processes**

**Problem:** How do we obtain a coherent view of the system?
When is it possible to order two events?

Three cases:

1. Events executed on the same process:
When is it possible to order two events?

Three cases:

1. Events executed on the same process:
   - if $e$ and $f$ are events on the same process and $e$ occurred before $f$, then $e$ happened-before $f$
When is it possible to order two events?

Three cases:

1. Events executed on the same process:
   - If $e$ and $f$ are events on the same process and $e$ occurred before $f$, then $e$ happened-before $f$

2. Communication events of the same message:
When is it possible to order two events?

Three cases:

1. Events executed on the same process:
   - if $e$ and $f$ are events on the same process and $e$ occurred before $f$, then $e$ happened-before $f$

2. Communication events of the same message:
   - if $e$ is the send event of a message and $f$ is the receive event of the same message, then $e$ happened-before $f$
When is it possible to order two events?

Three cases:

1. Events executed on the same process:
   - If \( e \) and \( f \) are events on the same process and \( e \) occurred before \( f \), then \( e \) happened-before \( f \).

2. Communication events of the same message:
   - If \( e \) is the send event of a message and \( f \) is the receive event of the same message, then \( e \) happened-before \( f \).

3. Events related by transitivity:
When is it possible to order two events?

Three cases:

1. Events executed on the same process:
   - if $e$ and $f$ are events on the same process and $e$ occurred before $f$, then $e$ happened-before $f$

2. Communication events of the same message:
   - if $e$ is the send event of a message and $f$ is the receive event of the same message, then $e$ happened-before $f$

3. Events related by transitivity:
   - if event $e$ happened-before event $g$ and event $g$ happened-before event $f$, then $e$ happened-before $f$
Happened-Before Relation

- Happened-before relation is denoted by $\rightarrow$
Happened-Before Relation

- Happened-before relation is denoted by →
- Illustration:

```
P_1  a  b  c
    P_2  d  e  f
    P_3  g  h  i
```
Happened-Before Relation

- Happened-before relation is denoted by →

Illustration:

- Events on the same process: examples: \( a \rightarrow b, \ a \rightarrow c, \ d \rightarrow f \)
Happened-Before Relation

- Happened-before relation is denoted by $\rightarrow$.

Illustration:

- Events on the same process:
  examples: $a \rightarrow b$, $a \rightarrow c$, $d \rightarrow f$

- Events of the same message:
  examples: $b \rightarrow e$, $f \rightarrow i$
Happened-Before Relation

- Happened-before relation is denoted by $\rightarrow$

**Illustration:**

- Events on the same process:
  examples: $a \rightarrow b$, $a \rightarrow c$, $d \rightarrow f$

- Events of the same message:
  examples: $b \rightarrow e$, $f \rightarrow i$

- Transitivity:
  examples: $a \rightarrow e$, $a \rightarrow i$, $e \rightarrow i$
Concurrent Events

- Events not related by happened-before relation
Concurrent Events

- Events not related by happened-before relation
- Concurrency relation is denoted by \( \ll || \)
Concurrent Events

- Events **not related** by happened-before relation
- Concurrency relation is denoted by \( || \)
- Illustration:
Concurrent Events

- Events **not related** by happened-before relation
- Concurrency relation is denoted by $||$
- Illustration:

```
P_1
  a --- b --- c

P_2
  d --- e --- f

P_3
  g --- h --- i
```

- **Examples:** $a || d$, $d || h$, $c || e$
Concurrent Events

- Events **not related** by happened-before relation
- Concurrency relation is denoted by $\parallel$
- Illustration:

![Diagram showing concurrently occurring events](image)

- **Examples:** $a \parallel d, \ d \parallel h, \ c \parallel e$
- Concurrency relation is **not transitive**: example: $a \parallel d$ and $d \parallel c$ but $a \not\parallel c$
Different Kinds of Clocks

- Logical Clocks
  - used to totally order all events

- Vector Clocks
  - used to track happened-before relation

- Matrix Clocks
  - used to track what other processes know about other processes

- Direct Dependency Clocks
  - used to track direct causal dependencies
Different Kinds of Clocks

- **Logical Clocks**
  - used to **totally order** all events

- **Vector Clocks**
  - used to track **happened-before** relation
Different Kinds of Clocks

- **Logical Clocks**
  - used to **totally order** all events

- **Vector Clocks**
  - used to track **happened-before** relation

- **Matrix Clocks**
  - used to track what other processes **know** about other processes
Different Kinds of Clocks

- **Logical Clocks**
  - used to **totally order** all events

- **Vector Clocks**
  - used to track **happened-before** relation

- **Matrix Clocks**
  - used to track what **other processes know** about other processes

- **Direct Dependency Clocks**
  - used to track **direct** causal dependencies
Logical Clock

- Implements the notion of **virtual time**
Logical Clock

- Implements the notion of **virtual time**
- Can be used to **totally order** all events
Logical Clock

- Implements the notion of virtual time
- Can be used to totally order all events
- Assigns timestamp to each event in a way that is consistent with the happened-before relation:

\[ e \rightarrow f \implies C(e) < C(f) \]

- \( C(e) \): timestamp for event \( e \)
- \( C(f) \): timestamp for event \( f \)
Implementing Logical Clock

- Each process has a local scalar clock, initialized to zero
  - $C_i$ denotes the local clock of process $P_i$
Implementing Logical Clock

- Each process has a local scalar clock, initialized to zero
  - $C_i$ denotes the local clock of process $P_i$

- Action depends on the type of the event:
  - Protocol for process $P_i$:
    - On executing an interval event $C_i := C_i + 1$
    - On sending a message $m$:
      - $C_i := C_i + 1$ piggyback $C_i$ on $m$
    - On receiving a message $m$:
      - let $t_m$ be the timestamp piggybacked on $m$
      - $C_i := \max(C_i; t_m) + 1$
Implementing Logical Clock

- Each process has a local *scalar* clock, initialized to zero
  - $C_i$ denotes the local clock of process $P_i$

- Action depends on the type of the event:

- Protocol for process $P_i$:

  - For an interval event:
    
    $C_i := C_i + 1$

  - For sending a message $m$:
    
    $C_i := C_i + 1$ piggyback $C_i$ on $m$

  - For receiving a message $m$:
    
    Let $t_m$ be the timestamp piggybacked on $m$
    
    $C_i := \max(f(C_i); t_m) + 1$
Implementing Logical Clock

- Each process has a local scalar clock, initialized to zero
  - $C_i$ denotes the local clock of process $P_i$

- Action depends on the type of the event:

- Protocol for process $P_i$:
  - On executing an interval event:
    \[
    C_i := C_i + 1
    \]
Implementing Logical Clock

- Each process has a local **scalar** clock, initialized to zero
  - $C_i$ denotes the local clock of process $P_i$

- Action depends on the type of the event:

- Protocol for process $P_i$:
  - On executing an **interval event**:
    $$C_i := C_i + 1$$
  - On **sending a message** $m$:
    $$C_i := C_i + 1$$
    piggyback $C_i$ on $m$
Implementing Logical Clock

- Each process has a local **scalar** clock, initialized to zero
  - \( C_i \) denotes the local clock of process \( P_i \)

- Action depends on the type of the event:

- Protocol for process \( P_i \):
  - On executing an **interval event**:
    \[
    C_i := C_i + 1
    \]
  - On **sending a message** \( m \):
    \[
    C_i := C_i + 1 \\
    \text{piggyback } C_i \text{ on } m
    \]
  - On **receiving a message** \( m \):
    - let \( t_m \) be the timestamp piggybacked on \( m \)
    \[
    C_i := \max\{C_i, t_m\} + 1
    \]
Implementing Logical Clock: An Illustration

\( a: \text{internal event} \quad C(a) = 1 \)
Implementing Logical Clock: An Illustration

\[ b \text{ and } c: \text{internal events} \quad C(b) = 1 \quad \text{and} \quad C(c) = 1 \]
Implementing Logical Clock: An Illustration

$d$: send event

$C(d) = 2$
Implementing Logical Clock: An Illustration

\[ e : \text{receive event} \quad C(e) = 3 \]
Implementing Logical Clock: An Illustration
Logical clock cannot be used to determine whether two events are concurrent.
Logical clock **cannot be used** to determine whether two events are concurrent.

\[ e \parallel f \text{ does not imply } C(e) = C(f) \]
Vector Clock

- Captures the **happened-before** relation
Vector Clock

- Captures the happened-before relation

- Assigns timestamp to each event such that:
  \[ e \rightarrow f \iff C(e) < C(f) \]

  \( C(e) \): timestamp for event \( e \)
  \( C(f) \): timestamp for event \( f \)
Comparing Two Vectors

- Vectors are compared **component-wise**: 

- Vectors are compared component-wise:
Vectors are compared component-wise:

Equality:

\[ V = V' \iff \forall i : V[i] = V'[i] \]
Vectors are compared **component-wise**:

- **Equality**:
  \[ V = V' \iff \forall i : V[i] = V'[i] \]

- **Less Than**:
  \[ V < V' \iff \forall i : V[i] \leq V'[i] \land \exists i : V[i] < V'[i] \]
Comparing Two Vectors

- Vectors are compared component-wise:
  - Equality:
    \[ V = V' \iff \langle \forall i : V[i] = V'[i] \rangle \]
  - Less Than:
    \[ V < V' \iff \langle \forall i : V[i] \leq V'[i] \rangle \land \langle \exists i : V[i] < V'[i] \rangle \]

\[
\begin{bmatrix}
1 \\
2 \\
0
\end{bmatrix}
<
\begin{bmatrix}
2 \\
3 \\
1
\end{bmatrix}
\quad\text{and}\quad
\begin{bmatrix}
2 \\
1 \\
1
\end{bmatrix}
<
\begin{bmatrix}
2 \\
3 \\
4
\end{bmatrix}
\quad\text{but}\quad
\begin{bmatrix}
1 \\
2 \\
1
\end{bmatrix}
\not<
\begin{bmatrix}
2 \\
1 \\
3
\end{bmatrix}
\]
Each process has a local vector clock

$C_i$ denotes the local clock of process $P_i$
Implementing Vector Clock

- Each process has a local vector clock
  - $C_i$ denotes the local clock of process $P_i$

- Action depends on the type of the event:
  - Protocol for process $P_i$:
    - On executing an interval event:
      $$C_i[i] := C_i[i] + 1$$
    - On sending a message $m$:
      $$C_i[i] := C_i[i] + 1$$
      piggyback $C_i$ on $m$
    - On receiving a message $m$:
      $$let t_m be the timestamp piggybacked on m;$$
      $$C_i[k] := max_{f \in C_i[k]} t_m;$$
      $$C_i[i] := C_i[i] + 1$$
Implementing Vector Clock

- Each process has a local vector clock
  - $C_i$ denotes the local clock of process $P_i$

- Action depends on the type of the event:

- Protocol for process $P_i$:
  
  - On executing an interval event:
    
    $C_i[i] := C_i[i] + 1$

  - On sending a message $m$:
    
    $C_i[i] := C_i[i] + 1$ piggyback $C_i$ on $m$

  - On receiving a message $m$:
    
    let $t_m$ be the timestamp piggybacked on $m$
    
    $C_i[k] := \max_{k} C_i[k]$; $t_m[k] := C_i[i] + 1$
Implementing Vector Clock

- Each process has a local vector clock
  - $C_i$ denotes the local clock of process $P_i$

- Action depends on the type of the event:

- Protocol for process $P_i$:
  - On executing an interval event:
    \[ C_i[i] := C_i[i] + 1 \]
Implementing Vector Clock

- Each process has a local vector clock
  - $C_i$ denotes the local clock of process $P_i$

- Action depends on the type of the event:

- Protocol for process $P_i$:
  - On executing an interval event:
    $$C_i[i] := C_i[i] + 1$$
  - On sending a message $m$:
    $$C_i[i] := C_i[i] + 1$$
    piggyback $C_i$ on $m$
Implementing Vector Clock

- Each process has a local vector clock
  - $C_i$ denotes the local clock of process $P_i$

- Action depends on the type of the event:

- Protocol for process $P_i$:
  - On executing an interval event:
    - $C_i[i] := C_i[i] + 1$
  - On sending a message $m$:
    - $C_i[i] := C_i[i] + 1$
    - piggyback $C_i$ on $m$
  - On receiving a message $m$:
    - let $t_m$ be the timestamp piggybacked on $m$
    - $\forall k \quad C_i[k] := \max\{C_i[k], t_m[k]\}$
    - $C_i[i] := C_i[i] + 1$
Implementing Vector Clock: An Illustration

\[ a: \text{internal event} \]

\[ C(a) = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \]
Implementing Vector Clock: An Illustration

\[ C(b) = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \quad \text{and} \quad C(c) = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \]

\(b\) and \(c\): internal events
Implementing Vector Clock: An Illustration

\[ C(d) = \begin{bmatrix} 2 \\ 0 \\ 0 \end{bmatrix} \]

d: send event
Implementing Vector Clock: An Illustration

\[ e: \text{receive event} \quad C(e) = \begin{bmatrix} 2 \\ 2 \\ 0 \end{bmatrix} \]
Implementing Vector Clock: An Illustration

- Inherent Limitations
- Ordering of Events
- Abstract Clocks
  - Different Kinds of Clocks
  - Logical Clock
  - Implementing Logical Clock
  - Implementing Logical Clock: An Illustration
  - Limitation of Logical Clock
  - Vector Clock
  - Comparing Two Vectors
  - Implementing Vector Clock
  - Implementing Vector Clock: An Illustration
  - Properties of Vector Clock
- Ordering of Messages
- State of a Distributed System
- Monitoring a Distributed System
How many comparisons are needed to determine whether an event $e$ happened-before another event $f$?

- Inherent Limitations
- Ordering of Events
- Abstract Clocks
  - Different Kinds of Clocks
  - Logical Clock
  - Implementing Logical Clock
  - Implementing Logical Clock: An Illustration
  - Limitation of Logical Clock
  - Vector Clock
  - Comparing Two Vectors
  - Implementing Vector Clock
  - Implementing Vector Clock: An Illustration
  - Properties of Vector Clock
- Ordering of Messages
- State of a Distributed System
- Monitoring a Distributed System
Properties of Vector Clock

■ How many comparisons are needed to determine whether an event \( e \) happened-before another event \( f \)?

❖ As many as \( N \) integers may need to be compared in the worst case, where \( N \) is the number of processes.
Properties of Vector Clock

- How many **comparisons** are needed to determine whether an event \( e \) happened-before another event \( f \)?
  - As many as \( N \) integers may need to be compared in the worst case, where \( N \) is the number of processes
  - Suppose \( e \) and \( f \) occurred on processes \( P_i \) and \( P_j \)
Properties of Vector Clock

How many comparisons are needed to determine whether an event \( e \) happened-before another event \( f \)?

- As many as \( N \) integers may need to be compared in the worst case, where \( N \) is the number of processes
- Suppose \( e \) and \( f \) occurred on processes \( P_i \) and \( P_j \)

\[
\begin{align*}
e &\rightarrow f \\
\text{if and only if}
\end{align*}
\]
Properties of Vector Clock

How many *comparisons* are needed to determine whether an event $e$ happened-before another event $f$?

- As many as $N$ integers may need to be compared in the worst case, where $N$ is the number of processes.
- Suppose $e$ and $f$ occurred on processes $P_i$ and $P_j$.

$$e \rightarrow f$$

if and only if

$$(i = j) \land (C(e)[i] < C(f)[i])$$
Properties of Vector Clock

How many comparisons are needed to determine whether an event $e$ happened-before another event $f$?

- As many as $N$ integers may need to be compared in the worst case, where $N$ is the number of processes.
- Suppose $e$ and $f$ occurred on processes $P_i$ and $P_j$

$$ e \rightarrow f $$

if and only if

$$ (i = j) \land (C(e)[i] < C(f)[i]) \lor (i \neq j) \land (C(e)[i] \leq C(f)[i]) $$
For many applications, messages should be delivered in certain order to be interpreted meaningfully.
For many applications, messages should be delivered in certain order to be interpreted meaningfully.

Example:

- $m_2$ cannot be interpreted until $m_1$ has been received.
- Tom receives $m_2$ before $m_1$: an undesirable behavior.
Useful Notations

For a message $m$:
- $\text{src}(m)$: source process of $m$
- $\text{dst}(m)$: destination process of $m$
- $\text{snd}(m)$: send event of $m$
- $\text{rcv}(m)$: send event of $m$
A message $w$ **causally precedes** a message $m$ if $\text{snd}(w) \rightarrow \text{snd}(m)$
Causal Delivery of Messages

■ A message \( w \) causally precedes a message \( m \) if:
\[ \text{snd}(w) \rightarrow \text{snd}(m) \]

■ An execution of a distributed system is said to be causally ordered if the following holds for every message \( m \):

\[ \text{every message that causally precedes } m \text{ and is destined for the same process as } m \text{ is delivered before } m \]
A message $w$ **causally precedes** a message $m$ if $snd(w) \rightarrow snd(m)$

An execution of a distributed system is said to be **causally ordered** if the following holds for every message $m$:

Every message that **causally precedes** $m$ and is destined for the same process as $m$ is delivered before $m$.

Mathematically, for every message $w$:

\[(snd(w) \rightarrow snd(m)) \land (dst(w) = dst(m)) \Rightarrow rcv(w) \rightarrow rcv(m)\]
A Causally Ordered Delivery Protocol

- Proposed by Birman, Schiper and Stephenson (BSS)

- Assumption:
  - Communication is broadcast based: a process sends a message to every other process.
  - Each process maintains a vector with one entry for each process:
    - Let \( V_i \) denote the vector for process \( P_i \).
    - The \( j \)th entry of \( V_i \) refers to the number of messages that \( P_i \) knows of that have been broadcast by process \( P_j \) that \( P_i \) knows of.
A Causally Ordered Delivery Protocol

- Proposed by Birman, Schiper and Stephenson (BSS)

- Assumption:
  - communication is **broadcast based**: a process sends a message to every other process
A Causally Ordered Delivery Protocol

- Proposed by Birman, Schiper and Stephenson (BSS)

- Assumption:
  - communication is **broadcast based**: a process sends a message to every other process

- Each process maintains a vector with one entry for each process:
  - let $V_i$ denote the vector for process $P_i$
  - the $j^{th}$ entry of $V_i$ refers to the number of messages that have been broadcast by process $P_j$ that $P_i$ knows of
The BSS Protocol

Protocol for process \( P_i \):

- On broadcasting a message \( m \):
  - piggyback \( V_i \) on \( m \):
    \[
    V_i[j] = V_i[j] + 1
    \]

- On arrival of a message \( m \) from process \( P_j \):
  - let \( V_m \) be the vector piggybacked on \( m \)
  - deliver \( m \) once \( V_i \cdot V_m \)

- On delivery of a message \( m \) sent by process \( P_j \):
  - \( V_i[j] = V_i[j] + 1 \)
The BSS Protocol

- Protocol for process $P_i$:
  - On broadcasting a message $m$:
    
    piggyback $V_i$ on $m$
    
    $V_i[i] := V_i[i] + 1$
The BSS Protocol

- Protocol for process $P_i$:
  - On broadcasting a message $m$:
    piggyback $V_i$ on $m$
    $V_i[i] := V_i[i] + 1$
  - On arrival of a message $m$ from process $P_j$:
    let $V_m$ be the vector piggybacked on $m$
    deliver $m$ once $V_i \geq V_m$
The BSS Protocol

- Protocol for process \( P_i \):
  - On broadcasting a message \( m \):
    - piggyback \( V_i \) on \( m \)
    
    \[
    V_i[i] := V_i[i] + 1
    \]
  - On arrival of a message \( m \) from process \( P_j \):
    - let \( V_m \) be the vector piggybacked on \( m \)
    - deliver \( m \) once \( V_i \geq V_m \)
  - On delivery of a message \( m \) sent by process \( P_j \):
    
    \[
    V_i[j] := V_i[j] + 1
    \]
The BSS Protocol: An Illustration
The BSS Protocol: An Illustration

The BSS Protocol: An Illustration

Inherent Limitations
Ordering of Events
Abstract Clocks
Ordering of Messages
❖ Ordering of Messages
❖ Useful Notations
❖ Causal Delivery of Messages
❖ A Causally Ordered Delivery Protocol
❖ The BSS Protocol
❖ The BSS Protocol: An Illustration
❖ Another Causally Ordered Delivery Protocol
❖ When to Deliver a Message?
❖ The SES Protocol
❖ The SES Protocol (Continued)
❖ The SES Protocol: An Illustration

State of a Distributed System
Monitoring a Distributed System
The BSS Protocol: An Illustration
The BSS Protocol: An Illustration

\[ P_1 \]
\[
\begin{bmatrix}
0 & 0 \\
0 & 0
\end{bmatrix}
\]

\[ P_2 \]
\[
\begin{bmatrix}
0 & 0 \\
0 & 0
\end{bmatrix}
\]

\[ P_3 \]
\[
\begin{bmatrix}
0 & 0 \\
0 & 0
\end{bmatrix}
\]
The BSS Protocol: An Illustration

Inherent Limitations
Ordering of Events
Abstract Clocks
Ordering of Messages
❖ Ordering of Messages
❖ Useful Notations
❖ Causal Delivery of Messages
❖ A Causally Ordered Delivery Protocol
❖ The BSS Protocol
❖ The BSS Protocol: An Illustration
❖ Another Causally Ordered Delivery Protocol
❖ When to Deliver a Message?
❖ The SES Protocol
❖ The SES Protocol (Continued)
❖ The SES Protocol: An Illustration

State of a Distributed System
Monitoring a Distributed System

Department of Computer Science, The University of Texas at Dallas
The BSS Protocol: An Illustration
The BSS Protocol: An Illustration
The BSS Protocol: An Illustration
Another Causally Ordered Delivery Protocol

- Proposed by Schiper, Eggli and Sandoz (SES)

- Assumption:
  - communication is point-to-point
  - processes are using vector clock algorithm

- To determine whether a message has arrived out of causal order:
  - let $V_m$ denote the vector timestamp of message $m$
  - let $V_i$ denote the vector clock of process $P_i$
  - Does there exist a message $w$ that causally precedes $m$ and destined for $P_i$ such that $V_i \geq V_w$?
Another Causally Ordered Delivery Protocol

- Proposed by Schiper, Eggli and Sandoz (SES)

- Assumption:
  - communication is point-to-point
  - processes are using vector clock algorithm

Proposed by Schiper, Eggli and Sandoz (SES)

Assumption:

- communication is point-to-point
- processes are using vector clock algorithm

Another Causally Ordered Delivery Protocol

Proposed by Schiper, Eggli and Sandoz (SES)

Assumption:

- communication is point-to-point
- processes are using vector clock algorithm
Another Causally Ordered Delivery Protocol

- Proposed by Schiper, Eggli and Sandoz (SES)

- Assumption:
  - communication is point-to-point
  - processes are using vector clock algorithm

- To determine whether a message has arrived out of causal order:
  - Let $V_m$ denote the vector timestamp of message $m$
  - Let $V_i$ denote the vector clock of process $P_i$
  - Does there exist a message $w$ that causally precedes $m$ and destined for $P_i$ such that $V_i > V_w$?
Another Causally Ordered Delivery Protocol

- Proposed by Schiper, Eggli and Sandoz (SES)

- Assumption:
  - communication is point-to-point
  - processes are using vector clock algorithm

- To determine whether a message has arrived out of causal order:
  - let $V_m$ denote the vector timestamp of message $m$
  - let $V_i$ denote the vector clock of process $P_i$
Another Causally Ordered Delivery Protocol

- Proposed by Schiper, Eggli and Sandoz (SES)

- Assumption:
  - communication is point-to-point
  - processes are using vector clock algorithm

- To determine whether a message has arrived out of causal order:
  - let $V_m$ denote the vector timestamp of message $m$
  - let $V_i$ denote the vector clock of process $P_i$

  Does there exist a message $w$ that causally precedes $m$ and destined for $P_i$ such that $V_i \not> V_w$?
When to Deliver a Message?

A message $m$ destined for process $P_i$ can be delivered if:

- for every message $w$ that causally precedes $m$ and destined for $P_i$:
  - $V_i > V_w$ or, equivalently
  - let $\text{past}(m; j)$ denote the set of all messages that causally precede $m$ and are destined for process $P_j$:
    - $V_i > \max_{w \in \text{past}(m; i)} V_w$
When to Deliver a Message?

- A message $m$ destined for process $P_i$ can be delivered if:
  - for every message $w$ that causally precedes $m$ and destined for $P_i$:
    $$ V_i > V_w $$
When to Deliver a Message?

A message $m$ destined for process $P_i$ can be delivered if:

- for every message $w$ that causally precedes $m$ and destined for $P_i$:

$$V_i > V_w$$

or, equivalently
When to Deliver a Message?

- A message \( m \) destined for process \( P_i \) can be delivered if:
  - for every message \( w \) that causally precedes \( m \) and destined for \( P_i \):
    \[
    V_i > V_w
    \]
  - or, equivalently
    \[
    V_i > \max_{w \in \text{past}(m, i)} \{ V_w \}
    \]
  - let \( \text{past}(m, j) \) denote the set of all messages that causally precede \( m \) and are destined for process \( P_j \):
Each process maintains a list of tuples with at most one tuple for every other process

- let $DL_i$ denote the list for process $P_i$
Each process maintains a list of tuples with at most one tuple for every other process

- let $DL_i$ denote the list for process $P_i$

The list is piggybacked on every message a process sends

- let $DL_m$ denote the list for message $m$
The SES Protocol

- Each process maintains a list of tuples with at most one tuple for every other process
  - let $DL_i$ denote the list for process $P_i$

- The list is piggybacked on every message a process sends
  - let $DL_m$ denote the list for message $m$
    - if $past(m, j) = \emptyset$, then $DL_m$ does not contain a tuple for process $P_j$
The SES Protocol

- Each process maintains a list of tuples with at most one tuple for every other process
  - let \( DL_i \) denote the list for process \( P_i \)

- The list is piggybacked on every message a process sends
  - let \( DL_m \) denote the list for message \( m \)
    - if \( \text{past}(m, j) = \emptyset \), then \( DL_m \) does not contain a tuple for process \( P_j \)
    - Otherwise, the tuple for process \( P_j \) is given by:
      \[
      (j, \max_{w \in \text{past}(m, j)} \{V_w\})
      \]
Protocol for process $P_i$:

- On sending a message $m$ to process $P_j$:
  - piggyback DL$_i$ on $m$
  - remove entry for $P_j$ from DL$_i$, if any
  - add $(j; V_m)$ to DL$_i$

- On arrival of a message $m$ from process $P_j$:
  - if DL$_m$ does not contain any tuple for $P_i$ then deliver $m$
  - else let $(j; V_i)$ be the tuple for $P_i$ in DL$_m$ deliver $m$ once $V_i > V_m$ endif

- On delivery of a message $m$:
  - merge DL$_i$ and DL$_m$
Protocol for process $P_i$:

- On sending a message $m$ to process $P_j$:
  - piggyback $DL_i$ on $m$
  - remove entry for $P_j$ from $DL_i$, if any
  - add $(j, V_m)$ to $DL_i$

- On arrival of a message $m$ from process $P_j$:
  - if $DL_m$ does not contain any tuple for $P_i$ then deliver $m$
  - else let $(j, V_m)$ be the tuple for $P_i$ in $DL_m$ deliver $m$ once $V_i > V_m$

- On delivery of a message $m$:
  - merge $DL_i$ and $DL_m$
The SES Protocol (Continued)

- Protocol for process $P_i$:
  - On sending a message $m$ to process $P_j$:
    - piggyback $DL_i$ on $m$
    - remove entry for $P_j$ from $DL_i$, if any
    - add $(j, V_m)$ to $DL_i$
  - On arrival of a message $m$ from process $P_j$:
    - if $DL_m$ does not contain any tuple for $P_i$ then deliver $m$
    - else
      - let $(j, V)$ be the tuple for $P_i$ in $DL_m$
      - deliver $m$ once $V_i > V$
    - endif
The SES Protocol (Continued)

Protocol for process \( P_i \):

- On sending a message \( m \) to process \( P_j \):
  piggyback \( DL_i \) on \( m \)
  remove entry for \( P_j \) from \( DL_i \), if any
  add \((j, V_m)\) to \( DL_i \)

- On arrival of a message \( m \) from process \( P_j \):
  if \( DL_m \) does not contain any tuple for \( P_i \) then
    deliver \( m \)
  else
    let \((j, V)\) be the tuple for \( P_i \) in \( DL_m \)
    deliver \( m \) once \( V_i > V \)
  endif

- On delivery of a message \( m \):
  merge \( DL_i \) and \( DL_m \)
The SES Protocol: An Illustration

\[ P_1 \]
\[
\begin{bmatrix}
0 \\
0 \\
0 \\
\end{bmatrix}
\]

\[ P_2 \]
\[
\begin{bmatrix}
0 \\
0 \\
0 \\
\end{bmatrix}
\]

\[ P_3 \]
\[
\begin{bmatrix}
0 \\
0 \\
0 \\
\end{bmatrix}
\]
The SES Protocol: An Illustration

Inherent Limitations
Ordering of Events
Abstract Clocks
Ordering of Messages
❖ Ordering of Messages
Useful Notations
❖ Causal Delivery of Messages
❖ A Causally Ordered Delivery Protocol
❖ The BSS Protocol
❖ The BSS Protocol: An Illustration
❖ Another Causally Ordered Delivery Protocol
❖ When to Deliver a Message?
❖ The SES Protocol
❖ The SES Protocol (Continued)
❖ The SES Protocol: An Illustration

State of a Distributed System
Monitoring a Distributed System
The SES Protocol: An Illustration
The SES Protocol: An Illustration

- \( P_1 \)
  - \( a \)
  - Inherent Limitations
  - Ordering of Events
  - Abstract Clocks
  - Ordering of Messages
  - Useful Notations
  - Causal Delivery of Messages
  - A Causally Ordered Delivery Protocol
  - The BSS Protocol
  - The BSS Protocol: An Illustration
  - Another Causally Ordered Delivery Protocol
  - When to Deliver a Message?
  - The SES Protocol
  - The SES Protocol (Continued)
  - The SES Protocol: An Illustration

- \( P_2 \)
  - \( b \)
  - \( c \)
  - \( \{ (3, \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}) \} \)
  - \( \{ (3, \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}), (2, \begin{bmatrix} 2 \\ 0 \\ 0 \end{bmatrix}) \} \)

- \( P_3 \)
  - \( \{ \} \)
  - \( \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \)
The SES Protocol: An Illustration

Inherent Limitations
Ordering of Events
Abstract Clocks
Ordering of Messages
❖ Ordering of Messages
❖ Useful Notations
❖ Causal Delivery of Messages
❖ A Causally Ordered Delivery Protocol
❖ The BSS Protocol
❖ The BSS Protocol: An Illustration
❖ Another Causally Ordered Delivery Protocol
❖ When to Deliver a Message?
❖ The SES Protocol
❖ The SES Protocol (Continued)
❖ The SES Protocol: An Illustration

State of a Distributed System
Monitoring a Distributed System
The SES Protocol: An Illustration

Diagram:

- **P1**
  - a
  - \{ (3, \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} ) \} \{ (3, \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} ), (2, \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} ) \} \{ (3, \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} ) \}

- **P2**
  - c
  - d
  - \{ (3, \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} ) \} \{ (3, \begin{bmatrix} 2 \\ 0 \\ 0 \end{bmatrix} ) \} \{ (3, \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} ) \}

- **P3**
  - e
  - g
  - \{ \} \{ \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \}

- States and messages are exchanged between processes a, b, c, d, e, and g.
The SES Protocol: An Illustration

P₁

\{ (3, \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}) \} \quad \{ (3, \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}), (2, \begin{bmatrix} 2 \\ 0 \\ 0 \end{bmatrix}) \} \quad \{ 2 \} \quad \{ (3, \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}) \} \quad a \quad b \quad c \quad d

P₂

\{ \} \quad \{ (3, \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}) \} \quad \{ (3, \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}) \} \quad \{ (3, \begin{bmatrix} 2 \\ 0 \\ 0 \end{bmatrix}) \} \quad c \quad d

P₃

\{ \} \quad \{ (3, \begin{bmatrix} 0 \\ 1 \\ 2 \end{bmatrix}) \} \quad g \quad f

Department of Computer Science, The University of Texas at Dallas

CS 6378: Advanced Operating Systems – Chapter 5: Theoretical Foundations
State of a distributed system is a collection of states of all its processes and channels:
State of a distributed system is a collection of states of all its processes and channels:

- a process state is given by the values of all variables on the process

State of a Distributed System
State of a Distributed System

- State of a distributed system is a collection of states of all its processes and channels:
  - a process state is given by the values of all variables on the process
  - a channel state is given by the set of messages in transit in the channel
State of a Distributed System

State of a distributed system is a collection of states of all its processes and channels:

- a process state is given by the values of all variables on the process
- a channel state is given by the set of messages in transit in the channel

- can be determined by examining states of the two processes it connects
State of a Distributed System

- State of a distributed system is a collection of states of all its processes and channels:
  - a process state is given by the values of all variables on the process
  - a channel state is given by the set of messages in transit in the channel
    - can be determined by examining states of the two processes it connects

- State of a process is called local state or local snapshot
  - the textbook refers to local state as cut event
  - cut event is not same as event
State of a Distributed System

- State of a distributed system is a collection of states of all its processes and channels:
  - a process state is given by the values of all variables on the process
  - a channel state is given by the set of messages in transit in the channel
  - can be determined by examining states of the two processes it connects

- State of a process is called local state or local snapshot
  - the textbook refers to local state as cut event
  - cut event is not same as event

- State of a distributed system is called global state or global snapshot
Processes **change** their states by executing events.
Events and Local States

- Processes change their states by executing events

Example:

- Process $P_1$ changes its state from $x = 0$ to $x = 2$ on executing event $a$.
- Process $P_2$ changes its state from $y = 1$ to $y = 3$ on executing event $d$. 
Processes change their states by executing events.

Example:

- Process $P_1$ changes its state from $x = 0$ to $x = 2$ on executing event $a$.
- Process $P_2$ changes its state from $y = 2$ to $y = 3$ on executing event $d$.
Events and Local States

- Processes change their states by executing events.

Example:

- Process $P_1$ changes its state from $x = 0$ to $x = 2$ on executing event $a$.
- Process $P_2$ changes its state from $y = 2$ to $y = 3$ on executing event $d$. 
When is a Global State Meaningful?

Example:

![Diagram showing two processes, P1 and P2, and their states for variables x and y.]

- For P1:
  - x = 0
  - y = 1

- For P2:
  - x = 2
  - y = 2

Is it possible for x to be 0 and y to be 3 at the same time? Does f; x; g form a meaningful global state?
When is a Global State Meaningful?

- Example:

```
\begin{align*}
  x &= 0 \\
  y &= 1 \\
  x &= 2 \\
  y &= 2 \\
  x &= 3 \\
  y &= 3
\end{align*}
```

Is it possible for $x$ to be 0 and $y$ to be 3 at the same time?
When is a Global State Meaningful?

Example:

- Is it possible for $x$ to be 0 and $y$ to be 3 at the same time?
- Does $\{s, x\}$ form a meaningful global state?
Revisiting Happened-Before Relation

- Typically, happened-before relation is defined on events
Typically, happened-before relation is defined on events.

The relation can be extended to local states as follows:

- Let $s$ be a local state of process $P_i$.
- Let $t$ be a local state of process $P_j$.
- $s \!\rightarrow\! t$ if there exist events $e$ and $f$ such that:
  1. $P_i$ executed $e$ after $s$,
  2. $P_j$ executed $f$ before $t$, and
  3. $e \!\rightarrow\! f$.
Revisiting Happened-Before Relation

- Typically, happened-before relation is defined on events.
- The relation can be extended to local states as follows:
  - let $s$ be a local state of process $P_i$
Typically, happened-before relation is defined on events.

The relation can be extended to local states as follows:

- let $s$ be a local state of process $P_i$
- let $t$ be a local state of process $P_j$

Typically, happened-before relation can be defined on events. The relation can be extended to local states as follows:

- let $s$ be a local state of process $P_i$
- let $t$ be a local state of process $P_j$
Typically, happened-before relation is defined on events.

The relation can be extended to local states as follows:

- let $s$ be a local state of process $P_i$
- let $t$ be a local state of process $P_j$
- $s \rightarrow t$ if there exist events $e$ and $f$ such that:
  1. $P_i$ executed $e$ after $s$,
  2. $P_j$ executed $f$ before $t$,
  3. $e \prec f$
Revisiting Happened-Before Relation

- Typically, happened-before relation is defined on events.
- The relation can be extended to local states as follows:
  - let $s$ be a local state of process $P_i$
  - let $t$ be a local state of process $P_j$
  - $s \rightarrow t$ if there exist events $e$ and $f$ such that:
    1. $P_i$ executed $e$ after $s$,
Typically, happened-before relation is defined on events.

The relation can be extended to local states as follows:

- let $s$ be a local state of process $P_i$
- let $t$ be a local state of process $P_j$
- $s ightarrow t$ if there exist events $e$ and $f$ such that:
  1. $P_i$ executed $e$ after $s$,
  2. $P_j$ executed $f$ before $t$, and

Revisiting Happened-Before Relation
Typically, happened-before relation is defined on events.

The relation can be extended to local states as follows:

- let \( s \) be a local state of process \( P_i \)
- let \( t \) be a local state of process \( P_j \)
- \( s \rightarrow t \) if there exist events \( e \) and \( f \) such that:
  1. \( P_i \) executed \( e \) after \( s \),
  2. \( P_j \) executed \( f \) before \( t \), and
  3. \( e \rightarrow f \)
A Consistent Global State

- For a global state $G$, let $G[i]$ refer to the local state of process $P_i$ in $G$.
A Consistent Global State

- For a global state $G$, let $G[i]$ refer to the local state of process $P_i$ in $G$

- A global state $G$ is meaningful or consistent if

$$\forall i, j : i \neq j : (G[i] \rightarrow G[j]) \land (G[j] \rightarrow G[i])$$
A Consistent Global State

- For a global state $G$, let $G[i]$ refer to the local state of process $P_i$ in $G$.

- A global state $G$ is meaningful or consistent if

$$\forall i, j : i \neq j : G[i] \parallel G[j]$$
Recording a Consistent Global Snapshot

- Proposed by Chandy and Lamport (CL)
Recording a Consistent Global Snapshot

- Proposed by Chandy and Lamport (CL)

- Assumptions and Features:
  - channels satisfy first-in-first-out (FIFO) property
Recording a Consistent Global Snapshot

- Proposed by Chandy and Lamport (CL)

- Assumptions and Features:
  - channels satisfy **first-in-first-out (FIFO) property**
  - channels are **not required to be bidirectional**

- Proposed by Chandy and Lamport (CL)

- Assumptions and Features:
  - channels satisfy **first-in-first-out (FIFO) property**
  - channels are **not required to be bidirectional**
Recording a Consistent Global Snapshot

- Proposed by Chandy and Lamport (CL)

Assumptions and Features:
- channels satisfy first-in-first-out (FIFO) property
- channels are not required to be bidirectional
- communication topology may not be fully connected

Proposed by Chandy and Lamport (CL)

Assumptions and Features:
- channels satisfy first-in-first-out (FIFO) property
- channels are not required to be bidirectional
- communication topology may not be fully connected

Proposed by Chandy and Lamport (CL)

Assumptions and Features:
- channels satisfy first-in-first-out (FIFO) property
- channels are not required to be bidirectional
- communication topology may not be fully connected

Proposed by Chandy and Lamport (CL)

Assumptions and Features:
- channels satisfy first-in-first-out (FIFO) property
- channels are not required to be bidirectional
- communication topology may not be fully connected
Recording a Consistent Global Snapshot

- Proposed by Chandy and Lamport (CL)

- Assumptions and Features:
  - channels satisfy first-in-first-out (FIFO) property
  - channels are not required to be bidirectional
  - communication topology may not be fully connected

- Messages exchanged by the underlying computation (whose snapshot is being recorded) are called application messages
Recording a Consistent Global Snapshot

- Proposed by Chandy and Lamport (CL)

- Assumptions and Features:
  - channels satisfy first-in-first-out (FIFO) property
  - channels are not required to be bidirectional
  - communication topology may not be fully connected

- Messages exchanged by the underlying computation (whose snapshot is being recorded) are called application messages

- Messages exchanged by the snapshot algorithm are called control messages
Initially: all processes are colored blue
- *Initially:* all processes are colored blue
- *Eventually:* all processes become red
The CL Protocol

- **Initially:** all processes are colored blue
- **Eventually:** all processes become red
- On changing color from blue to red:
  - record local snapshot
  - send a marker message along all outgoing channels
The CL Protocol

- **Initially**: all processes are colored **blue**
- **Eventually**: all processes become **red**
- **On changing color** from blue to red:
  
  - record local snapshot
  - send a marker message along all outgoing channels
- **On receiving marker message** along incoming channel $C$:
  
  - if color is blue then
    - change color from blue to red
  
  - endif
  
  - record state of channel $C$ as application messages received along $C$ since turning red
The CL Protocol

- **Initially**: all processes are colored *blue*
- **Eventually**: all processes become *red*

On **changing color** from blue to red:
- record local snapshot
- send a marker message along all outgoing channels

- On **receiving marker message** along incoming channel $C$:
  - if color is blue then
    - change color from blue to red
  - endif
  - record state of channel $C$ as application messages received along $C$ since turning red

- Any process can initiate the snapshot protocol by **spontaneously changing its color** from blue to red
The CL Protocol

- Initially: all processes are colored blue
- Eventually: all processes become red
- On changing color from blue to red:
  - record local snapshot
  - send a marker message along all outgoing channels
- On receiving marker message along incoming channel $C$:
  - if color is blue then
    - change color from blue to red
  - endif
  - record state of channel $C$ as application messages received along $C$ since turning red
- Any process can initiate the snapshot protocol by spontaneously changing its color from blue to red
  - there can be multiple initiators of the snapshot protocol
**Global Snapshot:** local snapshots of processes *just after they turn red*
Global Snapshot: local snapshots of processes just after they turn red

In-Transit Messages: blue application messages received by processes after they have turned red
The CL Protocol (Continued)

- Global Snapshot: local snapshots of processes *just after they turn red*

- In-Transit Messages: *blue* application messages received by processes *after they have turned red*

- Why is the global snapshot consistent?

Assume an application message has the same color as its sender.

Can a blue process receive a red application message?
The CL Protocol (Continued)

- **Global Snapshot**: local snapshots of processes *just after they turn red*

- **In-Transit Messages**: *blue* application messages received by processes *after they have turned red*

- Why is the global snapshot consistent?
  - Assume an application message has the same color as its sender

Global Snapshot: local snapshots of processes *just after they turn red*

In-Transit Messages: *blue* application messages received by processes *after they have turned red*

Why is the global snapshot consistent?

- Assume an application message has the same color as its sender
- **Global Snapshot**: local snapshots of processes *just after they turn red*

- **In-Transit Messages**: *blue* application messages received by processes *after they have turned red*

- Why is the global snapshot consistent?
  - Assume an application message has the same color as its sender
  - Can a blue process receive a red application message?
The CL Protocol: An Illustration

- Three processes: $P_1$, $P_2$ and $P_3$
- Five channels: $C_{1,2}$, $C_{1,3}$, $C_{2,1}$, $C_{2,3}$ and $C_{3,1}$
The CL Protocol: An Illustration (Continued)

1. All processes are blue
1. \( P_1 \) turns red
2. \( P_1 \) sends a marker message along \( C_{1,2} \) and \( C_{1,3} \)
1. $P_2$ receives the marker message along $C_{1,2}$ and turns red
2. $P_2$ sends a marker message along $C_{2,1}$ and $C_{2,3}$
3. $P_2$ records the state of $C_{1,2}$ as $\emptyset$
1. $P_3$ receives the marker message along $C_{1,3}$ and turns red
2. $P_3$ sends a marker message along $C_{3,1}$
3. $P_3$ records the state of $C_{1,3}$ as $\emptyset$
1. $P_1$ receives the marker message along $C_{2,1}$

2. $P_1$ records the state of $C_{2,1}$ as $\{m_4, m_5\}$
1. \( P_3 \) receives the marker message along \( C_{2,3} \)
2. \( P_3 \) records the state of \( C_{2,3} \) as \( \{m_1\} \)
1. $P_1$ receives the marker message along $C_{3,1}$
2. $P_1$ records the state of $C_{3,1}$ as $\{m_2, m_3\}$
Many distributed computations obey the following paradigm:

- A process is either in **active** state or **passive** state.
- A process can send an application message only when it is active.
- An active process can become passive at any time.
- A passive process, on receiving an application message, becomes active.

Intuitively:
- If a process is active, then it is doing some work.
- If a process is passive, then it is idle.
- An active process uses an application message to send a part of its work to another process.
A Subclass of Distributed Computation

Many distributed computations obey the following paradigm:

- A process is either in **active** state or **passive** state
Many distributed computations obey the following paradigm:

- A process is either in **active** state or **passive** state
- A process can send an application message *only when it is active*
A Subclass of Distributed Computation

Many distributed computations obey the following paradigm:

- A process is either in **active** state or **passive** state
- A process can send an application message *only when it is active*
- An active process can become passive at any time
A Subclass of Distributed Computation

Many distributed computations obey the following paradigm:

- A process is either in **active** state or **passive** state
- A process can send an application message *only when it is active*
- An active process can become passive at any time
- A passive process, on receiving an application message, becomes active
Many distributed computations obey the following paradigm:

- A process is either in active state or passive state
- A process can send an application message only when it is active
- An active process can become passive at any time
- A passive process, on receiving an application message, becomes active

Intuitively:
A Subclass of Distributed Computation

Many distributed computations obey the following paradigm:

- A process is either in **active** state or **passive** state
- A process can send an application message *only when it is active*
- An active process can become passive at any time
- A passive process, on receiving an application message, becomes active

Intuitively:

- if a process is active, then it is doing some work
A Subclass of Distributed Computation

Many distributed computations obey the following paradigm:

- A process is either in **active** state or **passive** state
- A process can send an application message *only when it is active*
- An active process can become passive at any time
- A passive process, on receiving an application message, becomes active

Intuitively:

- if a process is active, then it is doing some work
- if process is passive, then it is idle
A Subclass of Distributed Computation

- Many distributed computations obey the following paradigm:
  - A process is either in **active** state or **passive** state
  - A process can send an application message *only when it is active*
  - An active process can become passive at any time
  - A passive process, on receiving an application message, becomes active

- Intuitively:
  - if a process is active, then it is doing some work
  - if process is passive, then it is idle
  - an active process uses an application message to send a part of its work to another process
Distributed Computation: An Illustration

P₁

P₂

P₃
Inherent Limitations

Ordering of Events

Abstract Clocks

Ordering of Messages

State of a Distributed System

Monitoring a Distributed System

- A Subclass of Distributed Computation
- Distributed Computation: An Illustration
- Termination Detection
- Huang's Algorithm
- Huang's Algorithm (Continued)
- Huang's Algorithm: An Illustration

Distributed Computation: An Illustration
Distributed Computation: An Illustration

Inherent Limitations

Ordering of Events

Abstract Clocks

Ordering of Messages

State of a Distributed System

Monitoring a Distributed System

❖ A Subclass of Distributed Computation

❖ Distributed Computation: An Illustration

❖ Termination Detection

❖ Huang's Algorithm

❖ Huang's Algorithm (Continued)

❖ Huang's Algorithm: An Illustration

Department of Computer Science, The University of Texas at Dallas

CS 6378: Advanced Operating Systems – Chapter 5: Theoretical Foundations

Slide 40/44
Distributed Computation: An Illustration

- Inherent Limitations
- Ordering of Events
- Abstract Clocks
- Ordering of Messages
- State of a Distributed System
- Monitoring a Distributed System
- A Subclass of Distributed Computation
- Terminated Detection
- Huang's Algorithm
- Huang's Algorithm (Continued)
- Huang's Algorithm: An Illustration
Distributed Computation: An Illustration

- Inherent Limitations
- Ordering of Events
- Abstract Clocks
- Ordering of Messages
- State of a Distributed System
- Monitoring a Distributed System
  - A Subclass of Distributed Computation
  - Distributed Computation: An Illustration
  - Termination Detection
  - Huang's Algorithm
  - Huang's Algorithm (Continued)
  - Huang's Algorithm: An Illustration
Distributed Computation: An Illustration

P_1

m_1

P_2

m_2

m_3

P_3

m_4
Distributed Computation: An Illustration

- Inherent Limitations
- Ordering of Events
- Abstract Clocks
- Ordering of Messages
- State of a Distributed System
- Monitoring a Distributed System
  - A Subclass of Distributed Computation
  - Distributed Computation: An Illustration
  - Termination Detection
  - Huang's Algorithm
  - Huang's Algorithm (Continued)
  - Huang's Algorithm: An Illustration
Distributed Computation: An Illustration

\[ P_1 \rightarrow m_1 \rightarrow P_2 \rightarrow m_2 \rightarrow P_3 \rightarrow m_3 \rightarrow m_4 \rightarrow P_2 \rightarrow m_4 \rightarrow P_1 \]
Distributed Computation: An Illustration

- Ordering of Events
- Abstract Clocks
- Ordering of Messages
- State of a Distributed System
- Monitoring a Distributed System
- A Subclass of Distributed Computation
- Distributed Computation: An Illustration
- Termination Detection
- Huang’s Algorithm
- Huang’s Algorithm (Continued)
- Huang’s Algorithm: An Illustration

Diagram:

- **P_1**
  - Message **m_1**
  - Message **m_4**

- **P_2**
  - Message **m_2**

- **P_3**
  - Message **m_3**
Termination Detection

To detect if the computation has finished doing all the work

- To detect if the computation has finished doing all the work
- All processes have become passive
- All channels have become empty
- Different types of computations:
  - Diffusing: only one process is active in the beginning
  - Non-diffusing: any subset of processes can be active in the beginning
- No process knows which processes are active and which processes are passive

Huang's Algorithm

(Continued)

Huang's Algorithm: An Illustration
To detect if the computation has finished doing all the work
- all processes have become passive, and
Termination Detection

- To detect if the computation has finished doing all the work
  - all processes have become passive, and
  - all channels have become empty
Termination Detection

- To detect if the computation has finished doing all the work
  - all processes have become passive, and
  - all channels have become empty

- Different types of computations:
Termination Detection

To detect if the computation has finished doing all the work

- all processes have become passive, and
- all channels have become empty

Different types of computations:

- **diffusing**: only one process is active in the beginning
Termination Detection

- To detect if the computation has finished doing all the work
  - all processes have become passive, and
  - all channels have become empty

- Different types of computations:
  - **diffusing**: only one process is active in the beginning
  - **non-diffusing**: any subset of processes can be active in the beginning
Termination Detection

- To detect if the computation has finished doing all the work
  - all processes have become passive, and
  - all channels have become empty

- Different types of computations:
  - **diffusing**: only one process is active in the beginning
  - **non-diffusing**: any subset of processes can be active in the beginning
  - no process knows which processes are active and which processes are passive
Huang’s Algorithm

- **Assumption:** computation is diffusing
Huang’s Algorithm

- **Assumption**: computation is diffusing
  - the initially active process is called the **coordinator**
Huang’s Algorithm

- **Assumption:** computation is diffusing
  - the initially active process is called the **coordinator**
  - coordinator is responsible for detecting termination
Huang’s Algorithm

- **Assumption**: computation is diffusing
  - the initially active process is called the coordinator
    - coordinator is responsible for detecting termination

- **Initially**:
  - the coordinator has a weight of 1
  - all other processes have a weight of 0

- **Invariants**:
  - total amount of weight in the system is 1
  - a non-coordinator process has a non-zero weight if and only if it is active
  - a channel has a non-zero weight if and only if it is non-empty

- Weight of a channel is the sum of the weight of all its messages
Huang’s Algorithm

- **Assumption**: computation is diffusing
  - the initially active process is called the coordinator
    - coordinator is responsible for detecting termination

- **Initially**:
  - coordinator has a weight of 1
Huang’s Algorithm

- **Assumption:** computation is diffusing
  - the initially active process is called the **coordinator**
  - coordinator is responsible for detecting termination

- **Initially:**
  - coordinator has a weight of 1
  - all other processes have a weight of 0
Huang’s Algorithm

- **Assumption**: computation is diffusing
  - the initially active process is called the **coordinator**
  - coordinator is responsible for detecting termination

- **Initially**:
  - coordinator has a weight of 1
  - all other processes have a weight of 0

- **Invariants**:
Huang’s Algorithm

- **Assumption**: computation is diffusing
  - the initially active process is called the **coordinator**
  - coordinator is responsible for detecting termination

- **Initially**:
  - coordinator has a weight of 1
  - all other processes have a weight of 0

- **Invariants**:
  - total amount of weight in the system is 1
Huang’s Algorithm

- **Assumption**: computation is diffusing
  - the initially active process is called the **coordinator**
    - coordinator is responsible for detecting termination

- **Initially**:
  - coordinator has a weight of 1
  - all other processes have a weight of 0

- **Invariants**:
  - total amount of weight in the system is 1
  - a non-coordinator process has a non-zero weight *if and only if* it is active
**Huang's Algorithm**

- **Assumption**: computation is diffusing
  - the initially active process is called the **coordinator**
  - coordinator is responsible for detecting termination

- **Initially**:
  - coordinator has a weight of 1
  - all other processes have a weight of 0

- **Invariants**:
  - total amount of weight in the system is 1
  - a non-coordinator process has a non-zero weight *if and only if* it is active
  - a channel has a non-zero weight *if and only if* it is non-empty
Huang’s Algorithm

- **Assumption**: computation is diffusing
  - the initially active process is called the coordinator
  - coordinator is responsible for detecting termination

- **Initially**:
  - coordinator has a weight of 1
  - all other processes have a weight of 0

- **Invariants**:
  - total amount of weight in the system is 1
  - a non-coordinator process has a non-zero weight *if and only if* it is active
  - a channel has a non-zero weight *if and only if* it is non-empty
  - weight of a channel is the sum of the weight of all its messages
Huang’s Algorithm (Continued)

- **Actions:**
  - On sending an application message:
    - Send half of its weight along with the message.
  - On receiving an application message:
    - Add the weight of the message to the current weight.
  - On becoming passive:
    - Send the current weight to the coordinator.
  - Coordinator announces termination once:
    - It has become passive and
    - It has collected all the weight.
Huang’s Algorithm (Continued)

- **Actions:**
  - On **sending** an application message:
    - **On** sending an application message:
      - Send half of its weight along with the message.
    - **On** receiving an application message:
      - Add the weight of the message to the current weight.
    - **On** becoming passive:
      - Send the current weight to the coordinator.
  - Coordinator announces termination once:
    - It has become passive and
    - It has collected all the weight.
Huang’s Algorithm (Continued)

- **Actions:**
  - **On sending** an application message:
    - send half of its weight along with the message
Huang’s Algorithm (Continued)

Actions:

- On **sending** an application message:
  - send half of its weight along with the message

- On **receiving** an application message:
Huang’s Algorithm (Continued)

**Actions:**

- **On sending** an application message:
  - send half of its weight along with the message

- **On receiving** an application message:
  - add the weight of the message to the current weight
Huang’s Algorithm (Continued)

- **Actions:**
  - **On sending** an application message:
    - send half of its weight along with the message
  - **On receiving** an application message:
    - add the weight of the message to the current weight
  - **On becoming** passive:
Huang’s Algorithm ( Continued)

- **Actions:**
  - **On sending** an application message:
    - send half of its weight along with the message
  - **On receiving** an application message:
    - add the weight of the message to the current weight
  - **On becoming** passive:
    - send the current weight to the coordinator
Huang’s Algorithm (Continued)

Actions:

- **On sending** an application message:
  - send half of its weight along with the message

- **On receiving** an application message:
  - add the weight of the message to the current weight

- **On becoming** passive:
  - send the current weight to the coordinator

Coordinator announces termination once:
Huang’s Algorithm (Continued)

- **Actions:**
  - **On sending** an application message:
    - send half of its weight along with the message
  - **On receiving** an application message:
    - add the weight of the message to the current weight
  - **On becoming** passive:
    - send the current weight to the coordinator

- Coordinator announces termination once:
  - it has become passive and
Huang’s Algorithm (Continued)

Actions:

- **On sending** an application message:
  - send half of its weight along with the message

- **On receiving** an application message:
  - add the weight of the message to the current weight

- **On becoming** passive:
  - send the current weight to the coordinator

Coordinator announces termination once:

- it has become passive and
- it has collected all the weight
Huang’s Algorithm: An Illustration
Huang’s Algorithm: An Illustration

\[ m_1(\frac{1}{2}) \]

\[
\begin{align*}
P_1 & \quad \frac{1}{2} \\
0 & \quad \frac{1}{2} \\
0 & \quad 0
\end{align*}
\]
Huang’s Algorithm: An Illustration

\[ P_1 \quad m_1(\frac{1}{2}) \quad P_2 \]

\[ P_3 \]
Huang’s Algorithm: An Illustration

\[ P_1 \rightarrow m_1(\frac{1}{2}) \rightarrow P_2 \rightarrow m_2(\frac{1}{4}) \rightarrow P_3 \]
Huang’s Algorithm: An Illustration

\[ P_1 \quad m_1(\frac{1}{2}) \quad P_2 \quad m_2(\frac{1}{4}) \quad P_3 \]

\[ \begin{align*}
P_1 & : 1, \frac{1}{2} \\
P_2 & : 0, \frac{1}{2}, \frac{1}{4} \\
P_3 & : 0, \frac{1}{4} \end{align*} \]
Huang’s Algorithm: An Illustration

- Inherent Limitations
- Ordering of Events
- Abstract Clocks
- Ordering of Messages
- State of a Distributed System
- Monitoring a Distributed System
  - A Subclass of Distributed Computation
  - Distributed Computation: An Illustration
  - Termination Detection
  - Huang’s Algorithm
  - Huang’s Algorithm (Continued)
  - Huang’s Algorithm: An Illustration
Huang’s Algorithm: An Illustration

\[ P_1 \quad \frac{1}{2} \quad P_2 \quad m_1(\frac{1}{2}) \quad m_2(\frac{1}{4}) \quad m_3(\frac{1}{8}) \quad m_4(\frac{1}{16}) \quad 0 \]

\[ P_2 \quad \frac{1}{2} \quad \frac{1}{4} \quad \frac{3}{8} \quad 0 \]

\[ P_3 \quad \frac{1}{4} \quad \frac{1}{16} \quad 0 \]
Huang’s Algorithm: An Illustration

\begin{itemize}
  \item Inherent Limitations
  \item Ordering of Events
  \item Abstract Clocks
  \item Ordering of Messages
  \item State of a Distributed System
  \item Monitoring a Distributed System
    \begin{itemize}
      \item A Subclass of Distributed Computation
      \item Distributed Computation: An Illustration
      \item Termination Detection
    \end{itemize}
  \item Huang’s Algorithm
  \item Huang’s Algorithm (Continued)
  \item Huang’s Algorithm: An Illustration
\end{itemize}
Huang’s Algorithm: An Illustration

- Inherent Limitations
- Ordering of Events
- Abstract Clocks
- Ordering of Messages
- State of a Distributed System
- Monitoring a Distributed System
- A Subclass of Distributed Computation
- Distributed Computation: An Illustration
- Termination Detection
- Huang’s Algorithm
- Huang’s Algorithm (Continued)

Huang’s Algorithm: An Illustration

\[
\begin{align*}
P_1 & \quad 1 \quad \frac{1}{2} \quad \frac{9}{16} \quad \frac{15}{16} \\
P_2 & \quad \frac{1}{2} \quad \frac{1}{4} \quad \frac{3}{8} \quad 0 \\
P_3 & \quad \frac{1}{4} \quad \frac{1}{8} \quad \frac{1}{16} \quad 0
\end{align*}
\]
Huang’s Algorithm: An Illustration