Application of Event B to Global Causal Ordering for Fault Tolerant Transactions

Divakar Yadav and Michael Butler

Declarative Systems and Software Engineering
School of Electronics and Computer Science
University of Southampton
Southampton SO17 1BJ UK

Event B

- B Method is a proof based formal method developed by Abrial.

- Event B is event driven approach used together with B Method.

- Event B provides complete framework for developing mathematical model of distributed algorithms by

  - Rigorous description of problem.
  - Gradually introducing solution in refinement steps.
  - Verification of correctness of solution by discharging proof obligations.

- Atelier B, Click‘n’Prove , B Toolkit provides support for discharge of proof obligation through automatic and interactive prover.

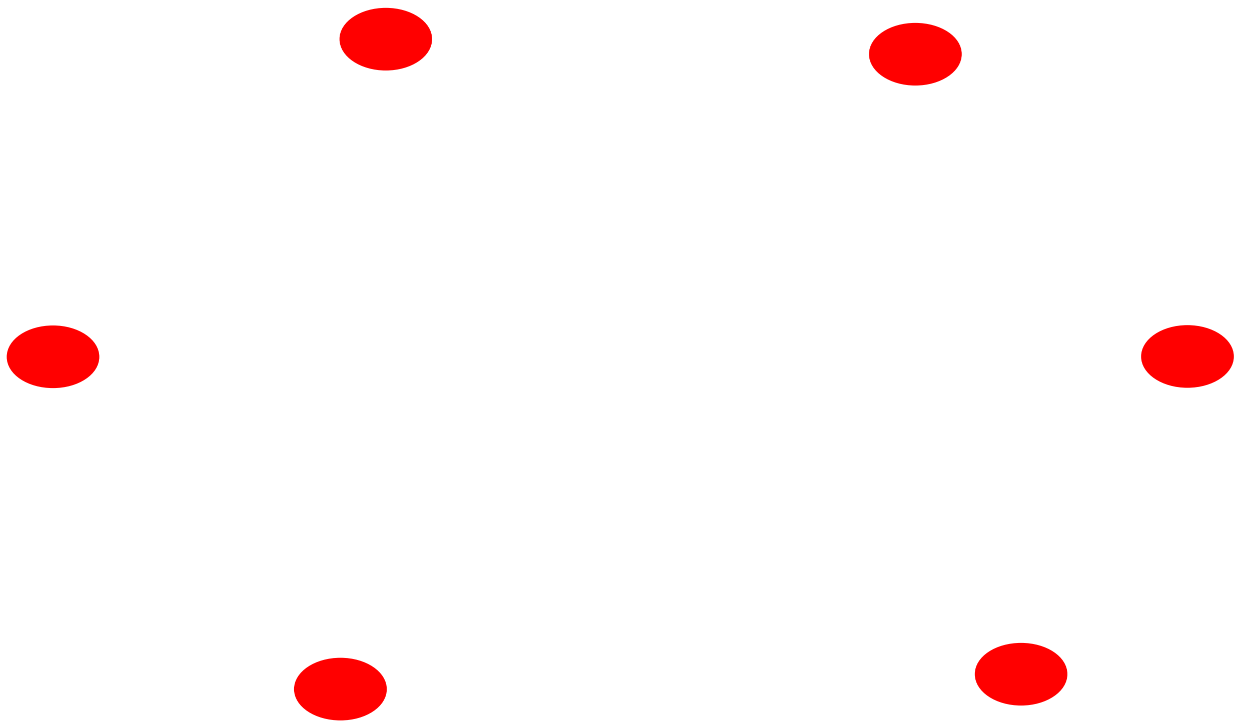
Fault Tolerant Transactions

Some issues on our ongoing work

- Distributed System is a collection of *autonomous computers* spatially separated.
- *Fragmentation* and *Replication* of data is a key issue in distributed database.
- *Synchronous replication* techniques require that all replica are updated before updating distributed transaction commits.
- *Read One Write All* (ROWA) based synchronous replication requires transaction to read one copy and write all copies.
- Fault Tolerance may be achieved by either *masking failures* or by following *well defined behaviour* suitable for recovery.

Synchronous Replication

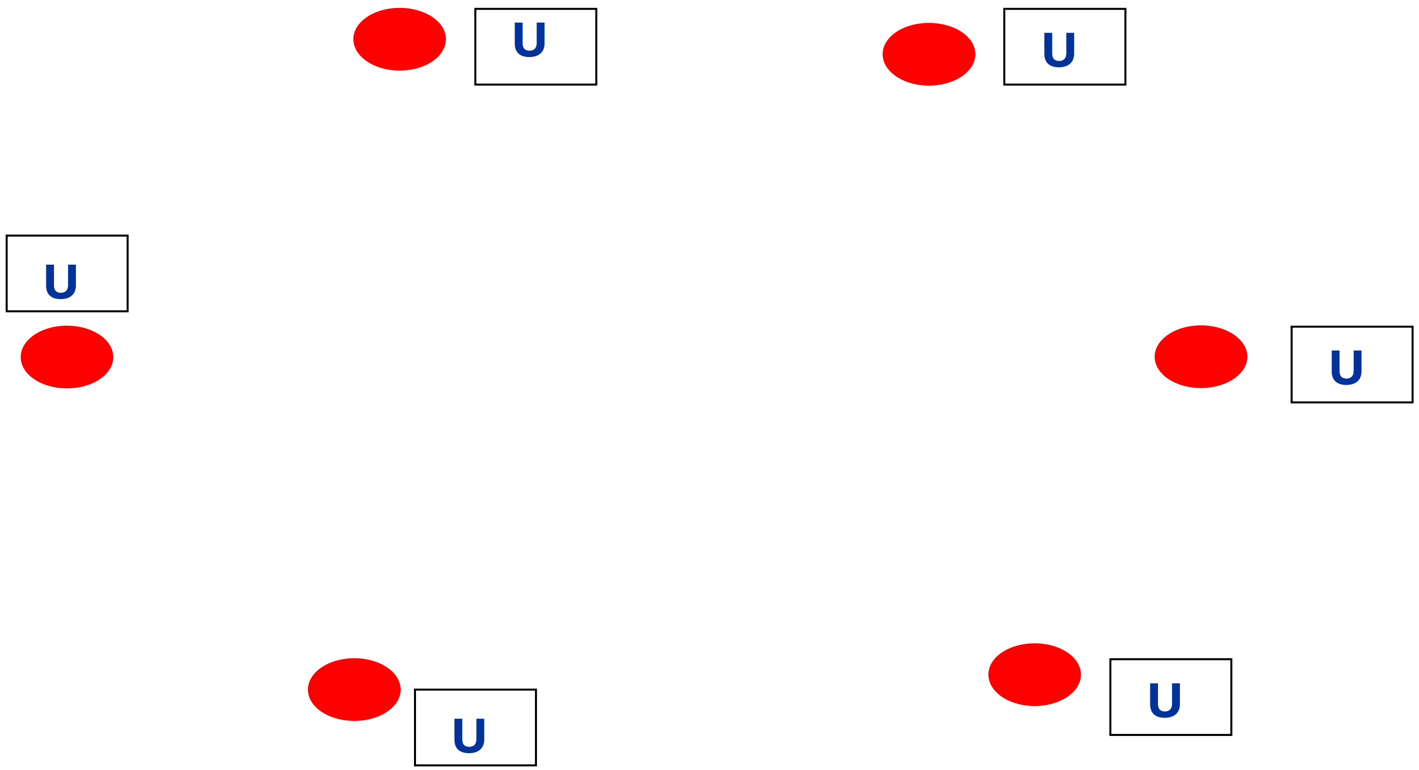
Read One Write All (ROWA)



- Sites contains the replica of data object.

Synchronous Replication

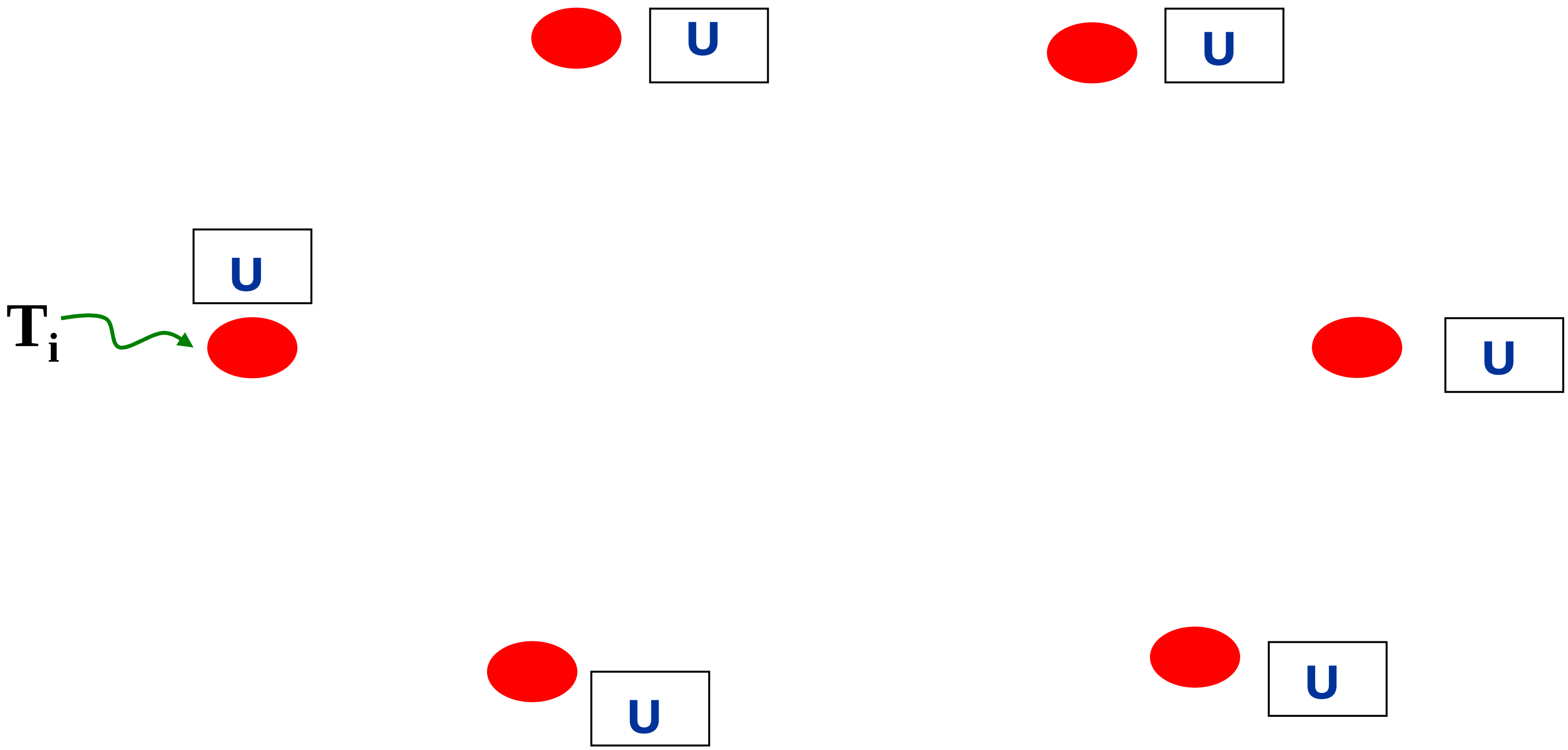
Read One Write All (ROWA)



- Initial value of data is U.

Synchronous Replication

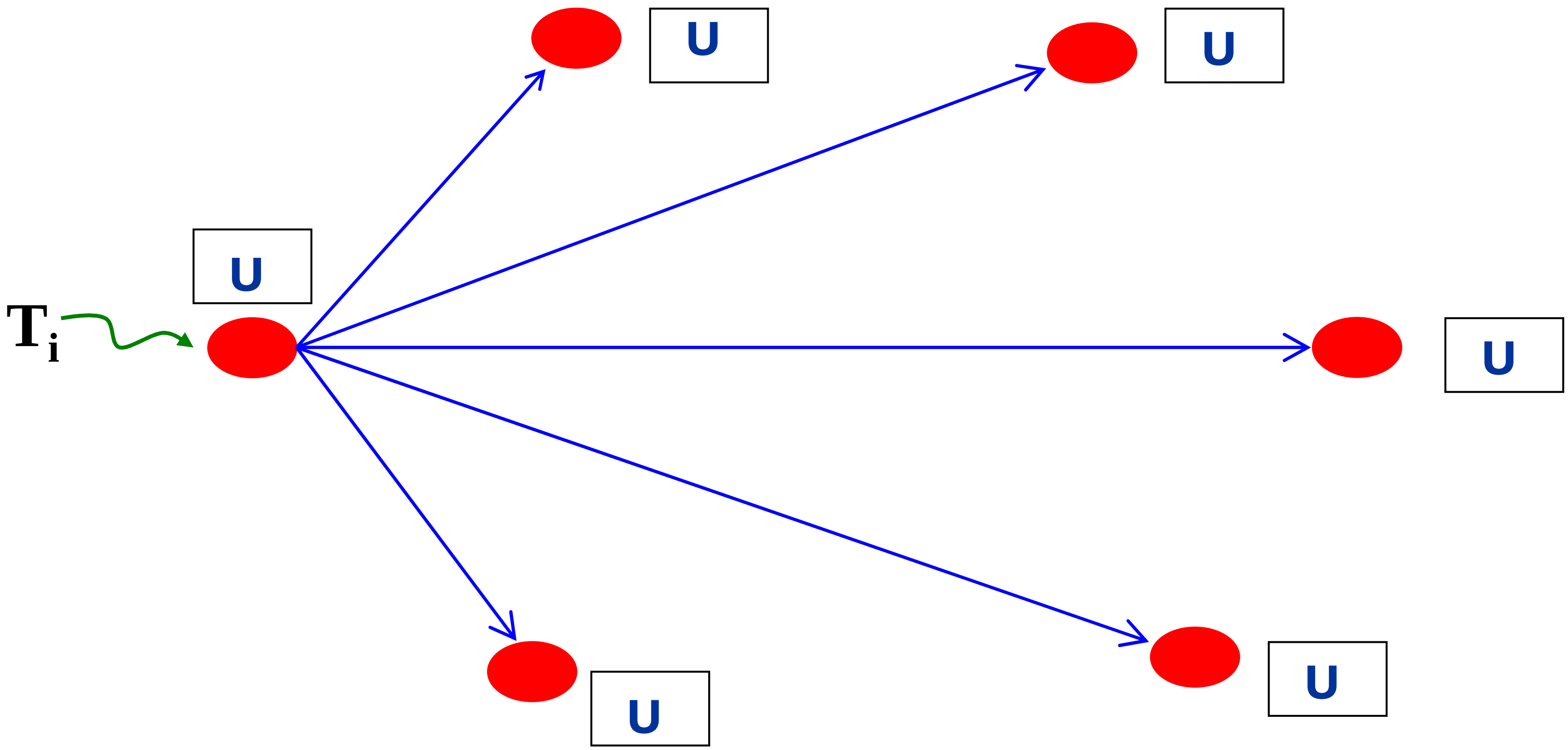
Read One Write All (ROWA)



- Transaction T_i is submitted at site S_i

Synchronous Replication

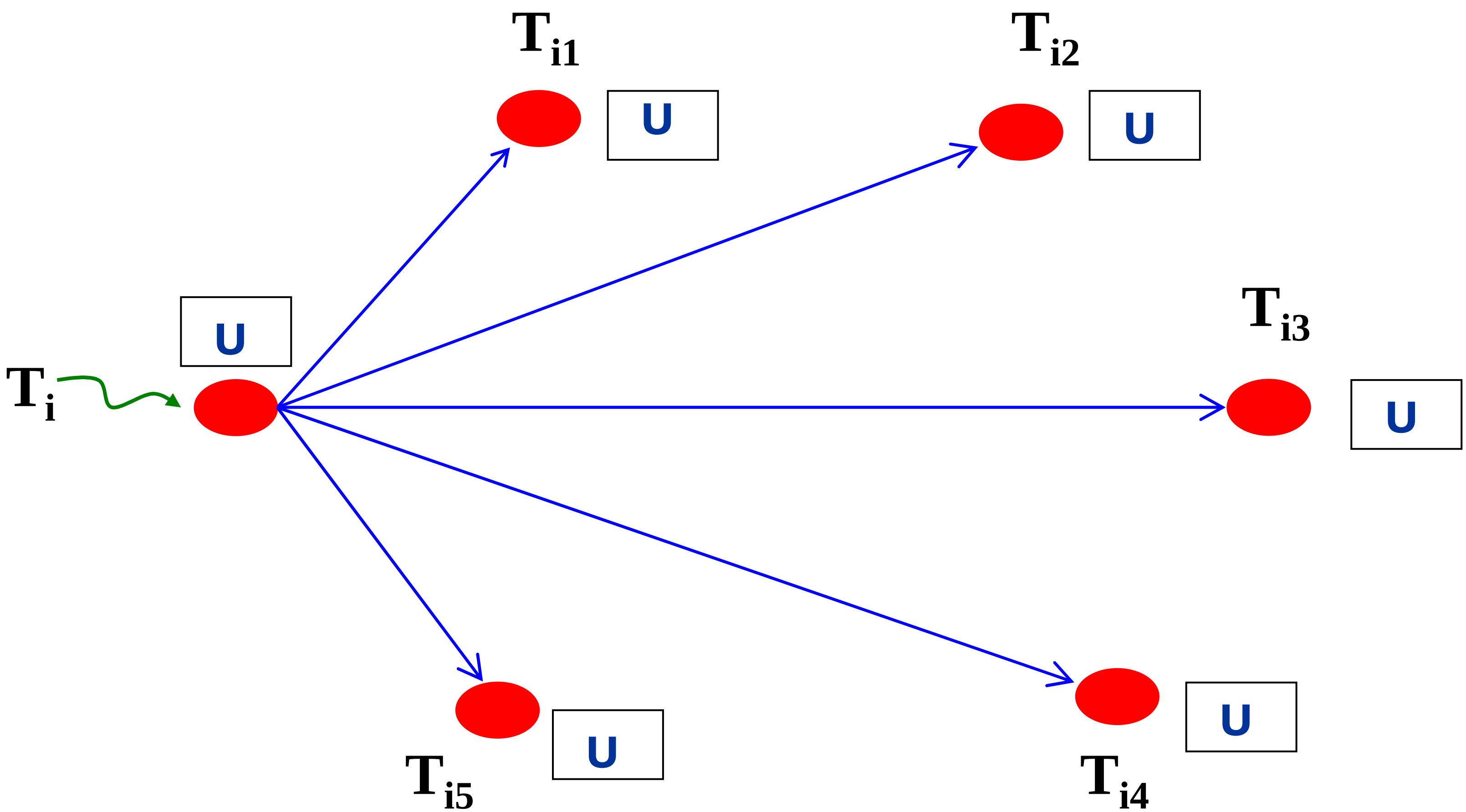
Read One Write All (ROWA)



- Site S_i sends messages to participating sites.

Synchronous Replication

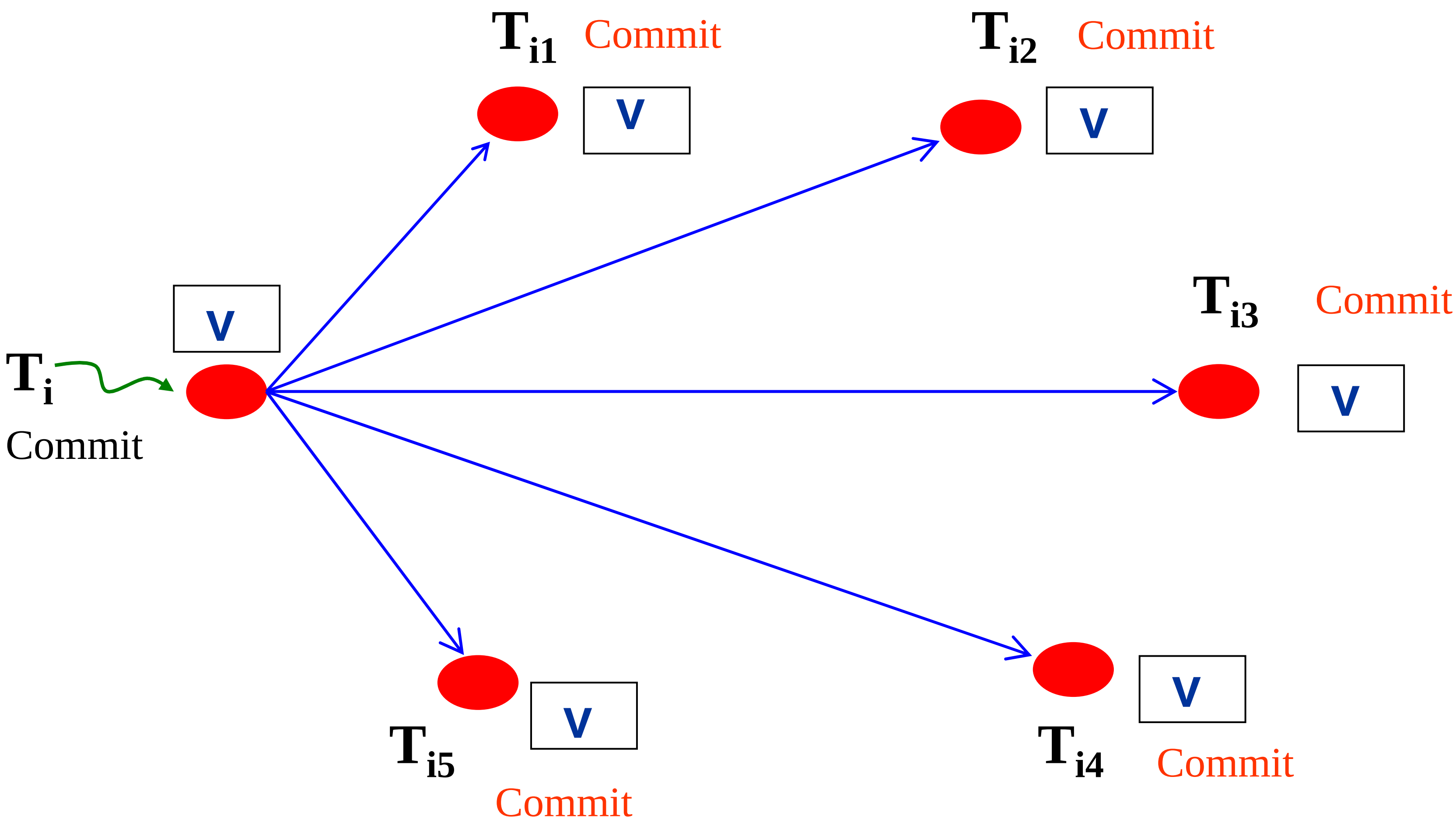
Read One Write All (ROWA)



- Sub transactions of T_i starts at participating sites

Synchronous Replication

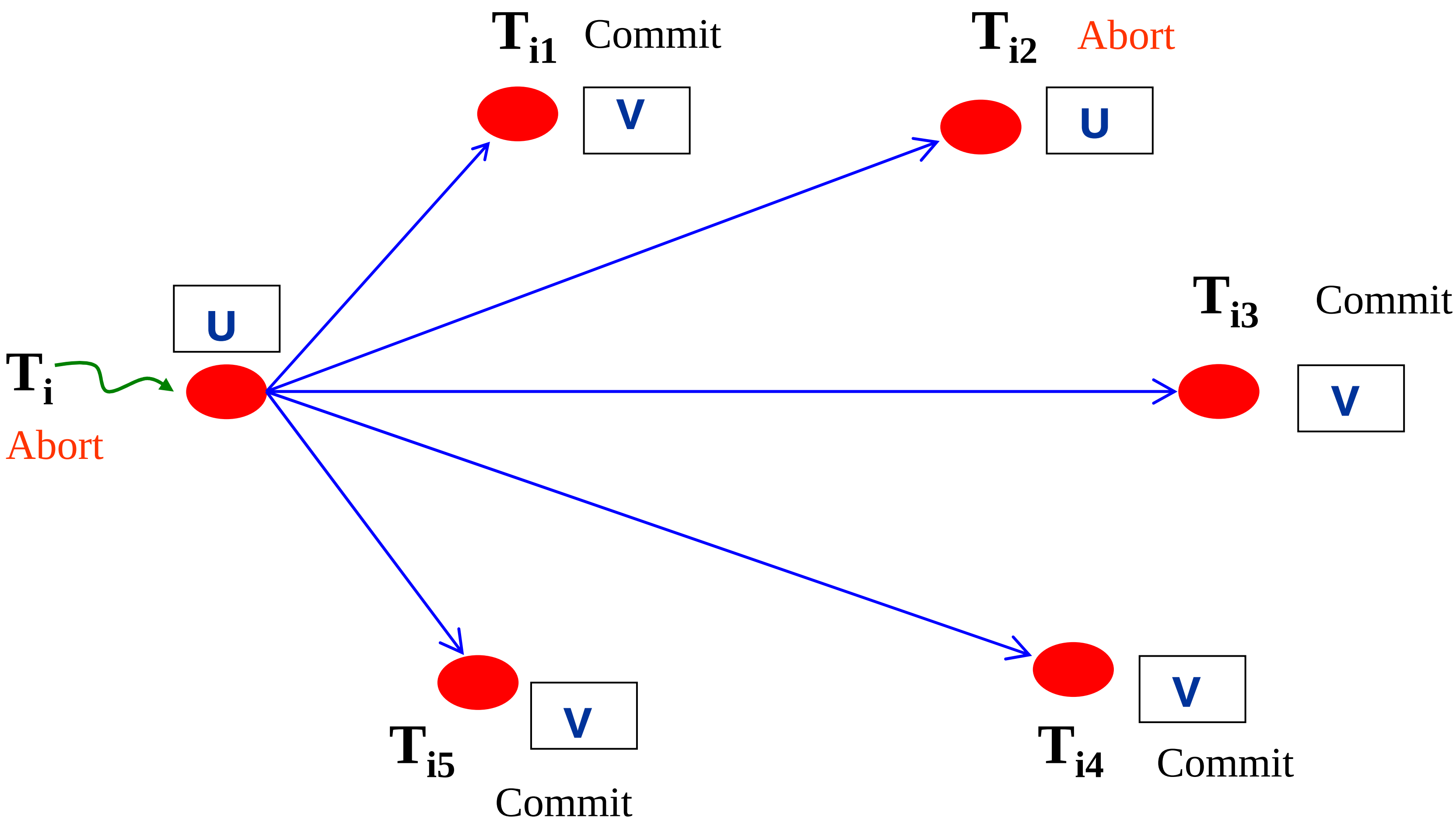
Read One Write All (ROWA)



- Distributed Transaction T_i commits only if all Sub transactions commits.

Synchronous Replication

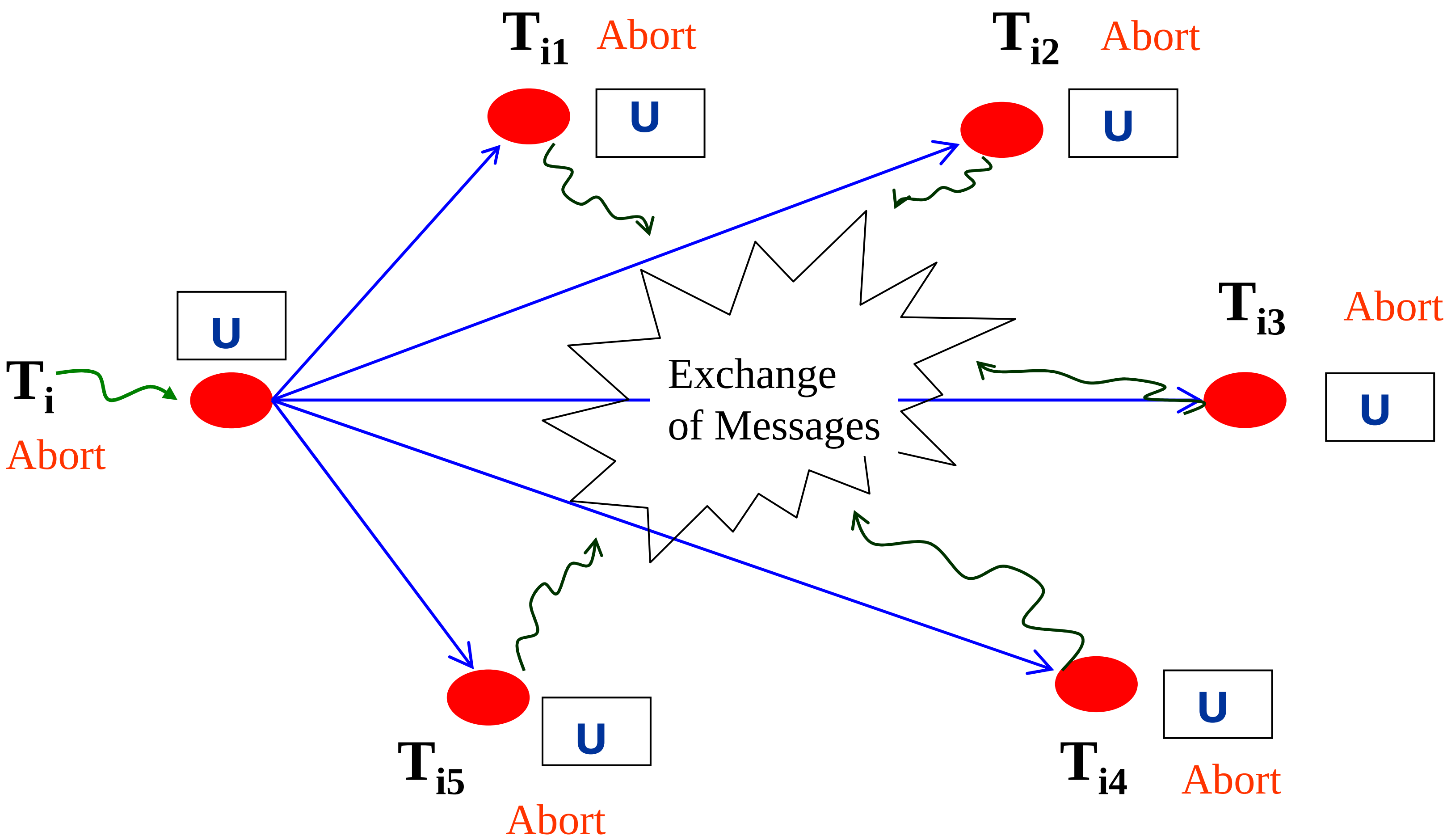
Read One Write All (ROWA)



- Distributed Transaction T_i Aborts if ***Any*** Sub transactions aborts.

Synchronous Replication

Read One Write All (ROWA)

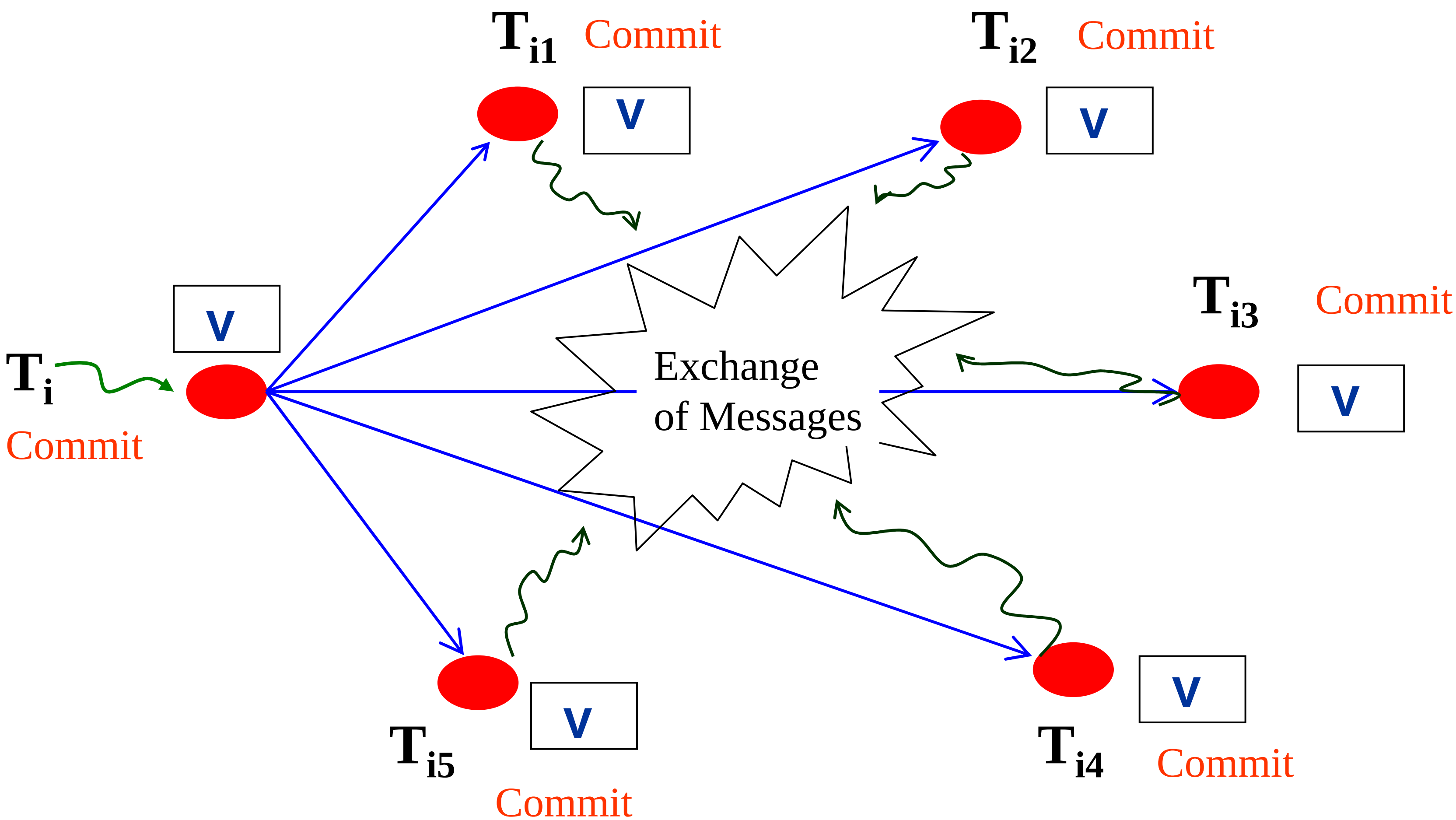


- If Distributed Transaction T_i Aborts, it aborts at ***all*** sites.

  $\Longrightarrow$ None of replica is updated.

Synchronous Replication

Read One Write All (ROWA)



- If Distributed Transaction T_i Commits, it commits at ***all*** sites.

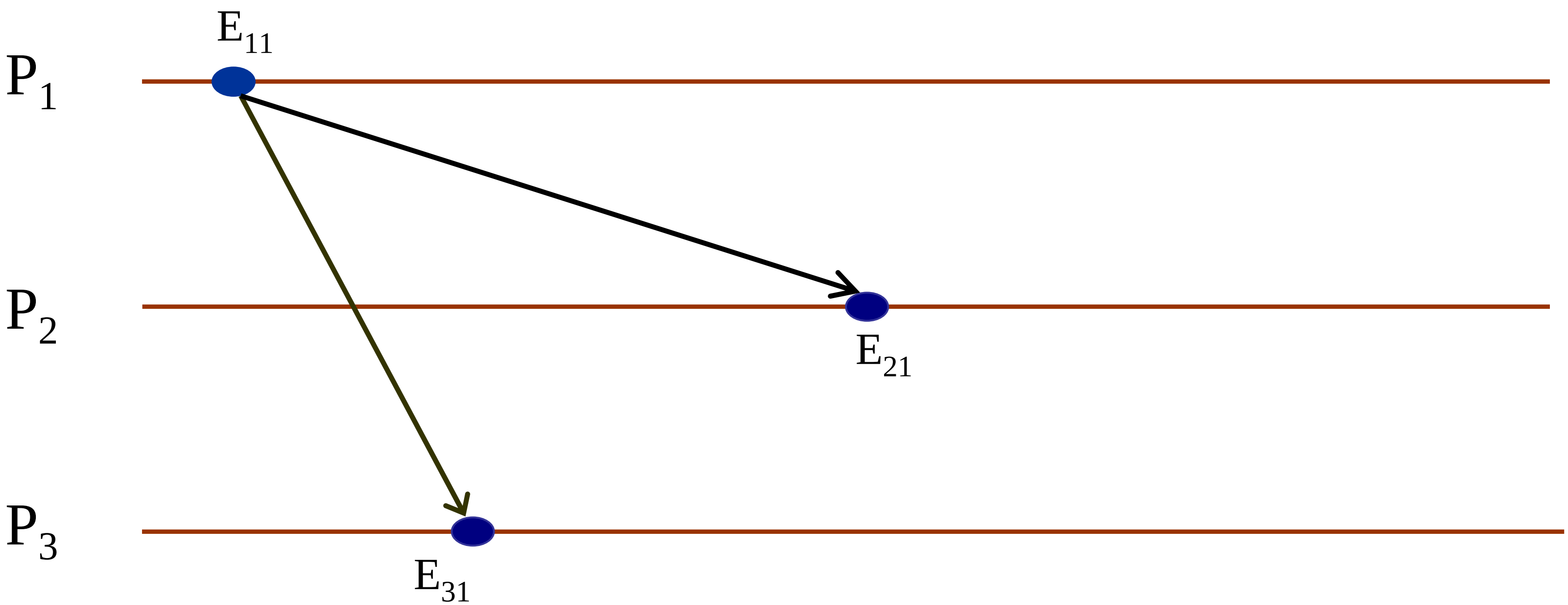
  $\Longrightarrow$ All replicas are updated.

From now onwards….

Application of Event B to

- Broadcast messaging system.
- Buffering of messages.
- Abstract model of causal order.
- Globally ordered delivery of messages.
- Implementation through vector clocks.

Broadcast Messaging



☞ Some Observations

- ❑ Processes communicate by broadcasting of messages.
- ❑ No loss or duplication of message.
- ❑ Messages are delivered after arbitrary delays.

Broadcast Messaging

SETS PROCESS; MESSAGE

VARIABLES sender , receive

INITIALISATION

sender := ∅ || receive := ∅

INVARIANT

sender ∈ MESSAGE ⇸ PROCESS

receive ∈ PROCESS ↔ MESSAGE

(p ↦m) ∈ receive ⇒ m ∈ dom(sender)

(p ↦m) ∈ receive ⇒ p ≠ sender(m))

OPERATIONS

Send(pp,mm) ≙

SELECT mm ∉ dom(sender)
THEN
sender := sender ∪ {mm ↦ pp}
END;

Receive (pp,mm) ≙

SELECT mm ∈ dom(sender)
∧ (pp ↦ mm) ∉ receive
∧ pp ≠sender(mm)
THEN
receive := receive ∪ {pp ↦ mm}
END

Happened Before Relation

- The ***happened before*** relation captures ***causal dependency*** between various events occurring in a process.

- ***Message Send*** and ***Message Receive*** are message events.

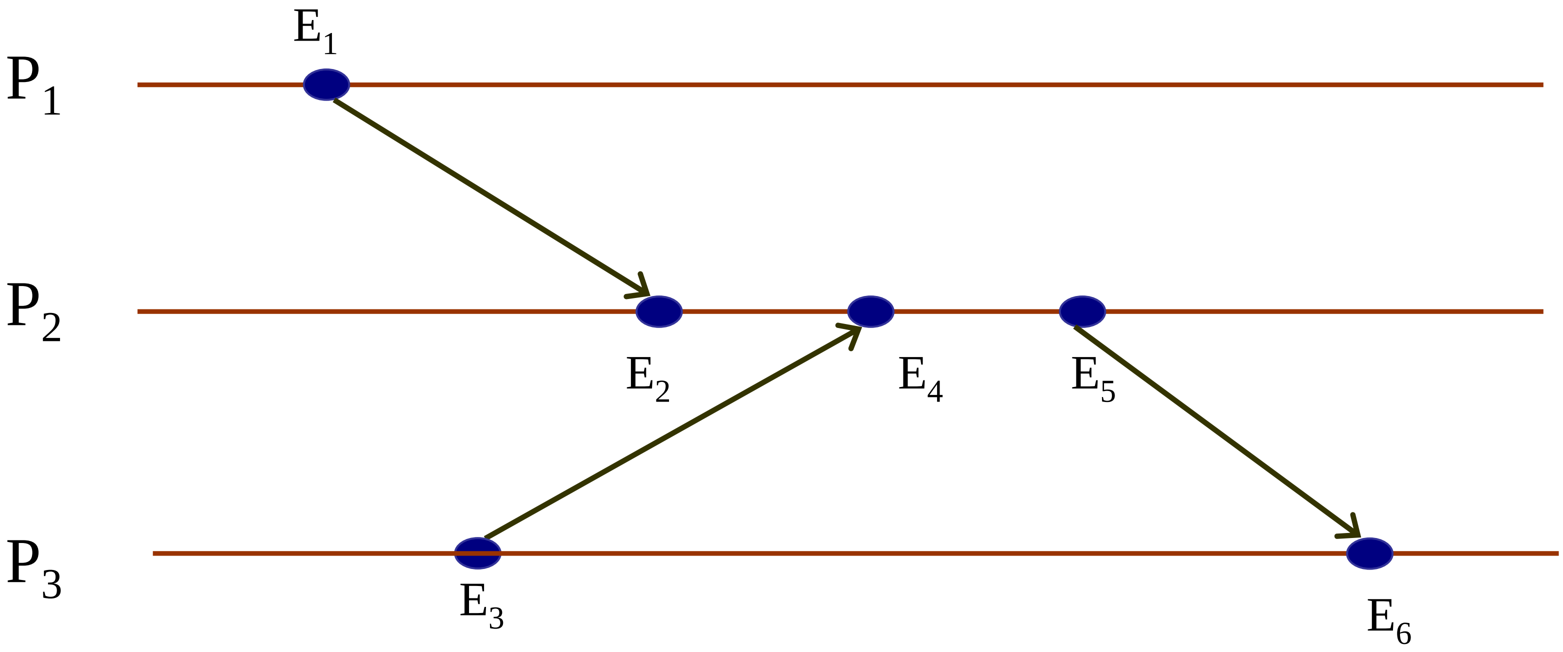
- Event A and B are ***causally related*** if either $A \rightarrow B$ or $B \rightarrow A$.

- Event A and B are ***concurrent*** (A || B) if $A \nrightarrow B$ and $B \nrightarrow A$.

- ***Transitivity*** : $A \rightarrow B \wedge B \rightarrow C \Longrightarrow A \rightarrow C$

Events Ordering

Some Observations

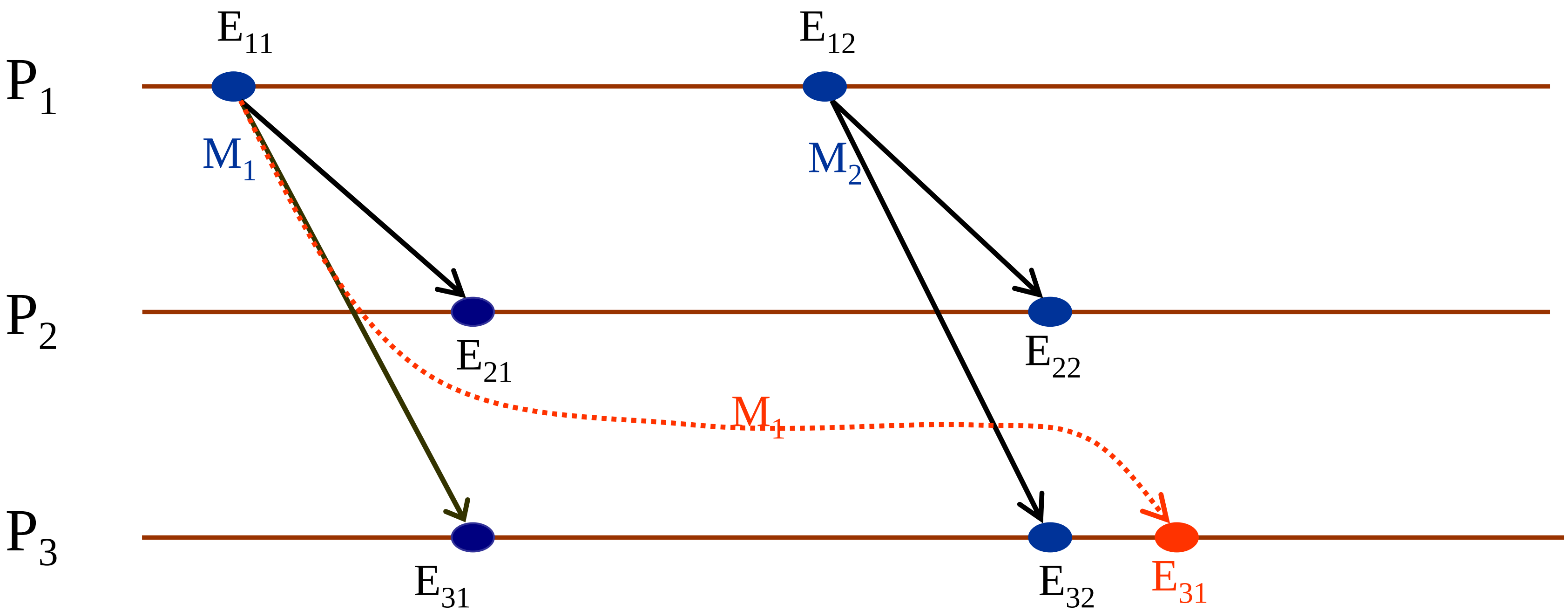


☞ Some Observations

- ✓ $\mathbf{E_1 \rightarrow E_2}$
- ✓ $\mathbf{E_2 \rightarrow E_4}$
- ✓ $\mathbf{E_1 \rightarrow E_2 \wedge E_2 \rightarrow E_4 \Longrightarrow E_1 \rightarrow E_4}$
- ✓ $\mathbf{E_1 \parallel E_3}$

Causal Ordering of Messages

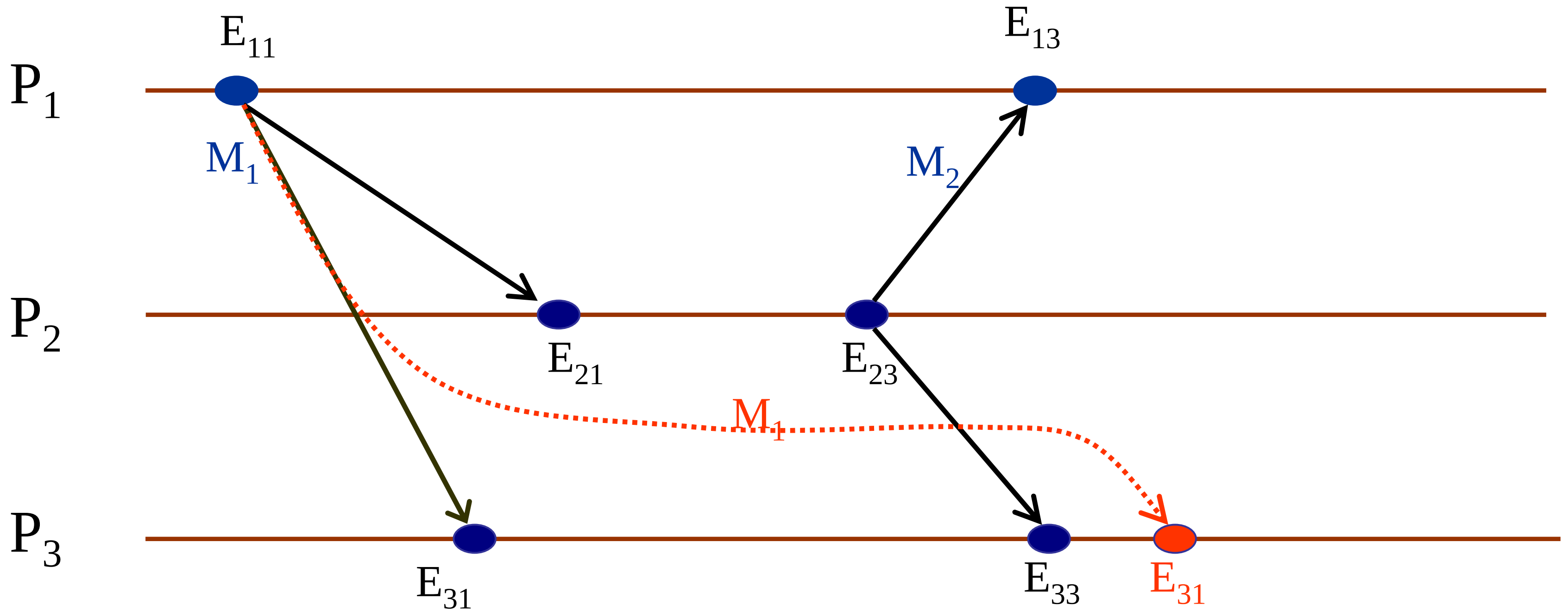
Some Observations



- Message M_1 ( ) shows violation of global causal ordering.

Causal Ordering of Messages

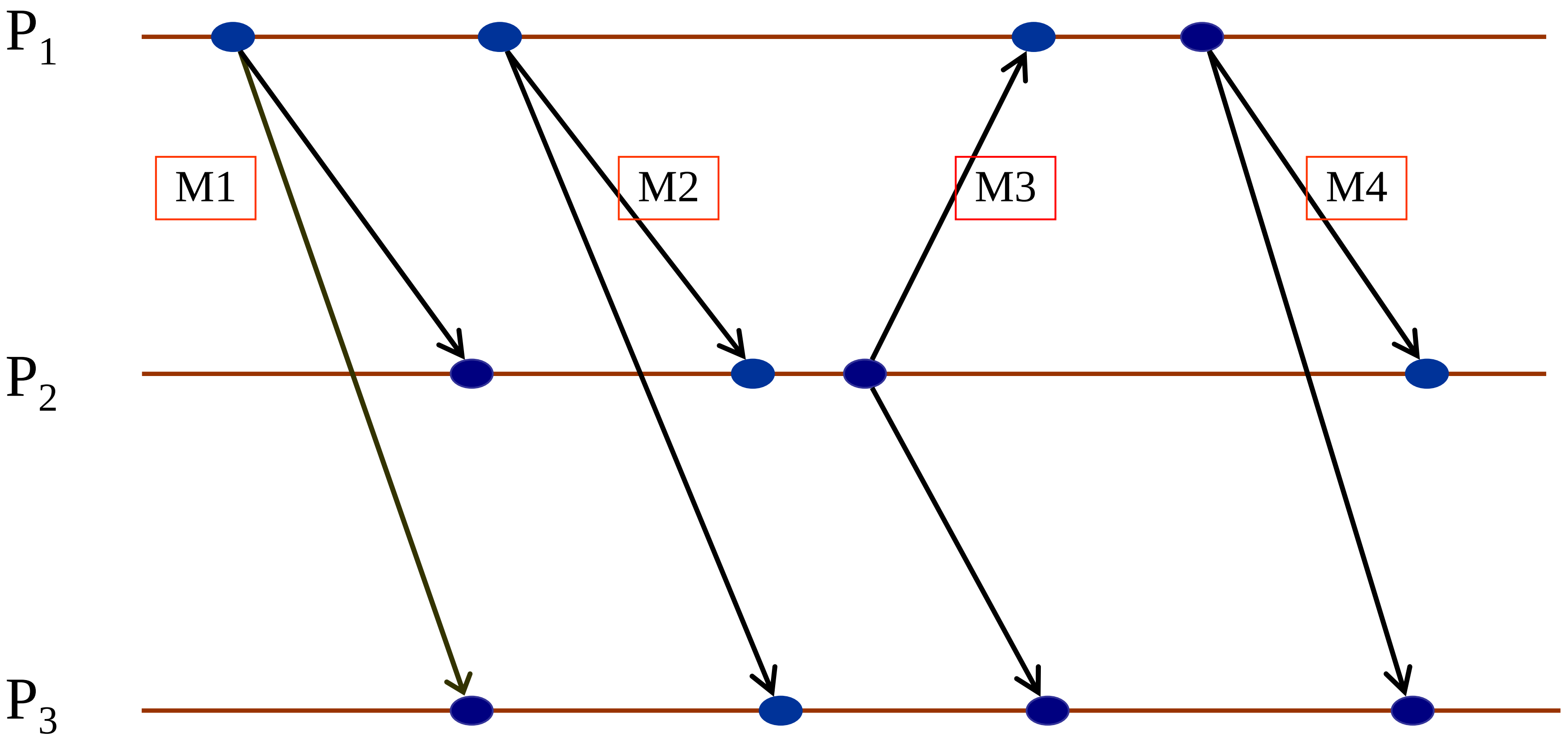
Some Observations



- Message M_1 ( ) shows violation of global causal ordering.

Causal Ordering of Messages

to Broadcast System



☞ Some Observations

$M1 \rightarrow M2$

$M2 \rightarrow M3$

$M1 \rightarrow M2 \wedge M2 \rightarrow M3 \Longrightarrow M1 \rightarrow M3$

Abstract Model of Causal Order

First Refinement

VARIABLES **sender , receive , order**

INVARIANT

order $\in$ MESSAGE $\leftrightarrow$ MESSAGE

If M1 $\rightarrow$ M2 and P has received M2, then P must have received M1

If M1 $\rightarrow$ M2 and P has sent M2, then P must have sent or received M1

order is transitive

INITIALISATION **sender := $\varnothing$ || receive := $\varnothing$ || order := $\varnothing$**

Abstract Model of Causal Order

First Refinement

VARIABLES **sender , receive , order**

INVARIANT

$$order \in MESSAGE \leftrightarrow MESSAGE$$

$$(m1 \mapsto m2) \in order \wedge (p \mapsto m2) \in receive \wedge p \neq sender(m1) \Rightarrow (p \mapsto m1) \in receive)$$

$$(m1 \mapsto m2) \in order \wedge (m2 \mapsto p) \in sender \Rightarrow ((m1 \mapsto p) \in sender \vee (p \mapsto m1) \in receive)$$

$$(m1 \mapsto m2) \in order \wedge (m2 \mapsto m3) \in order \Rightarrow (m1 \mapsto m3) \in order$$

INITIALISATION **sender := ∅ || receive := ∅ || order := ∅**

Operations

OPERATIONS

Send (pp,mm) $\triangleq$ **SELECT** mm $\notin$ dom(sender)
THEN
order := order $\cup$ ((sender~[{pp}] * {mm}) $\cup$ (receive[{pp}] * {mm}))
|| sender := sender $\cup$ {mm $\mapsto$ pp}

END;

Receive (pp,mm) $\triangleq$ **SELECT** mm $\in$ dom(sender)
$\wedge$ (pp $\mapsto$ mm) $\notin$ receive
$\wedge$ pp $\neq$ sender(mm)
$\wedge$ $\forall$m.(m $\in$ MESSAGE $\wedge$ (m $\mapsto$ mm) $\in$ order
$\wedge$ pp $\neq$ sender(m) $\Rightarrow$ (pp $\mapsto$ m) $\in$ receive)
THEN
receive := receive $\cup$ {pp $\mapsto$ mm}
END

END

Buffering of Messages

Second Refinement

- To ensure ***globally ordered delivery*** of messages at a recipient process, early message need be buffered.

- For any two message M1, M2 where M1 is ordered before M2 (M1→M2), If M2 ***arrives*** early at a process then M2 is ***buffered*** until M1 is received.

Buffering of Messages

Second Refinement

SETS **PROCESS ; MESSAGE** **VARIABLES** **sender , receive , order ,buffer**

INITIALISATION **sender := ∅ || receive := ∅ || order := ∅ || buffer := ∅**

Introducing a new event ***Arrive***

INVARIANT

buffer ∈ PROCESS ↔ MESSAGE

∧ ran(buffer) ⊆ dom (sender)

∧ ran(receive) ∩ ran(buffer) = ∅

OPERATIONS

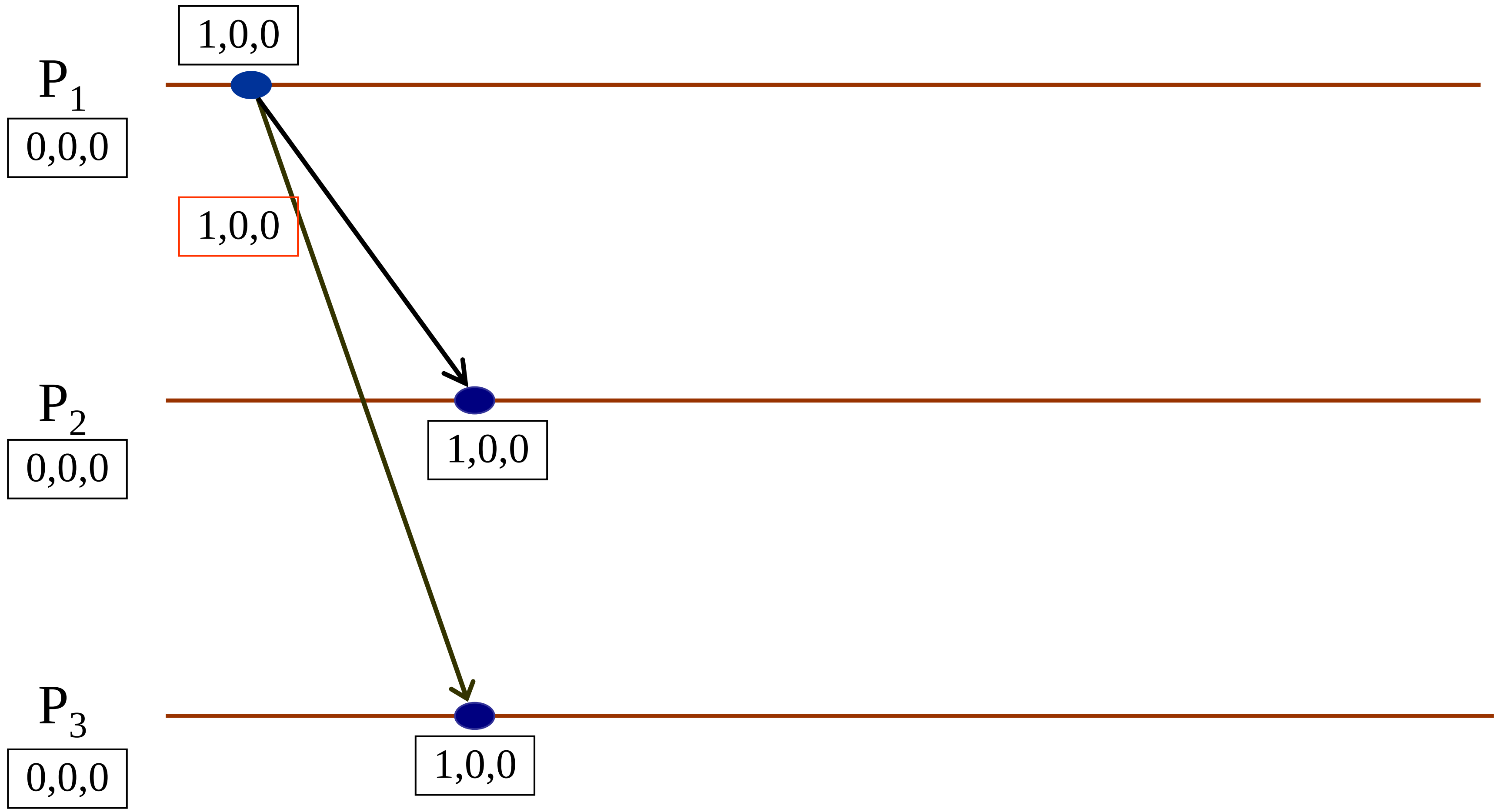
Arrive (pp,mm) ≙ SELECT mm ∈ dom(sender)
∧ (pp ↦ mm) ∉ buffer
∧ (pp ↦ mm) ∉ receive
∧ pp ≠ sender(mm)
THEN
buffer := buffer ∪ {pp ↦ mm}
END ;

Logical Clocks : Vector Clock

- Vector Clock uses a vector of Integers of size N, where N is number of processes in system.
- Process P_i maintains a vector clock VT_i.
- VT_i [i] is process P_i's own logical time.
- VT_i [j] is process P_i's best knowledge of time at process P_j.
- Proposed by Fidge and Mattern and based on Lamport's scalar clocks

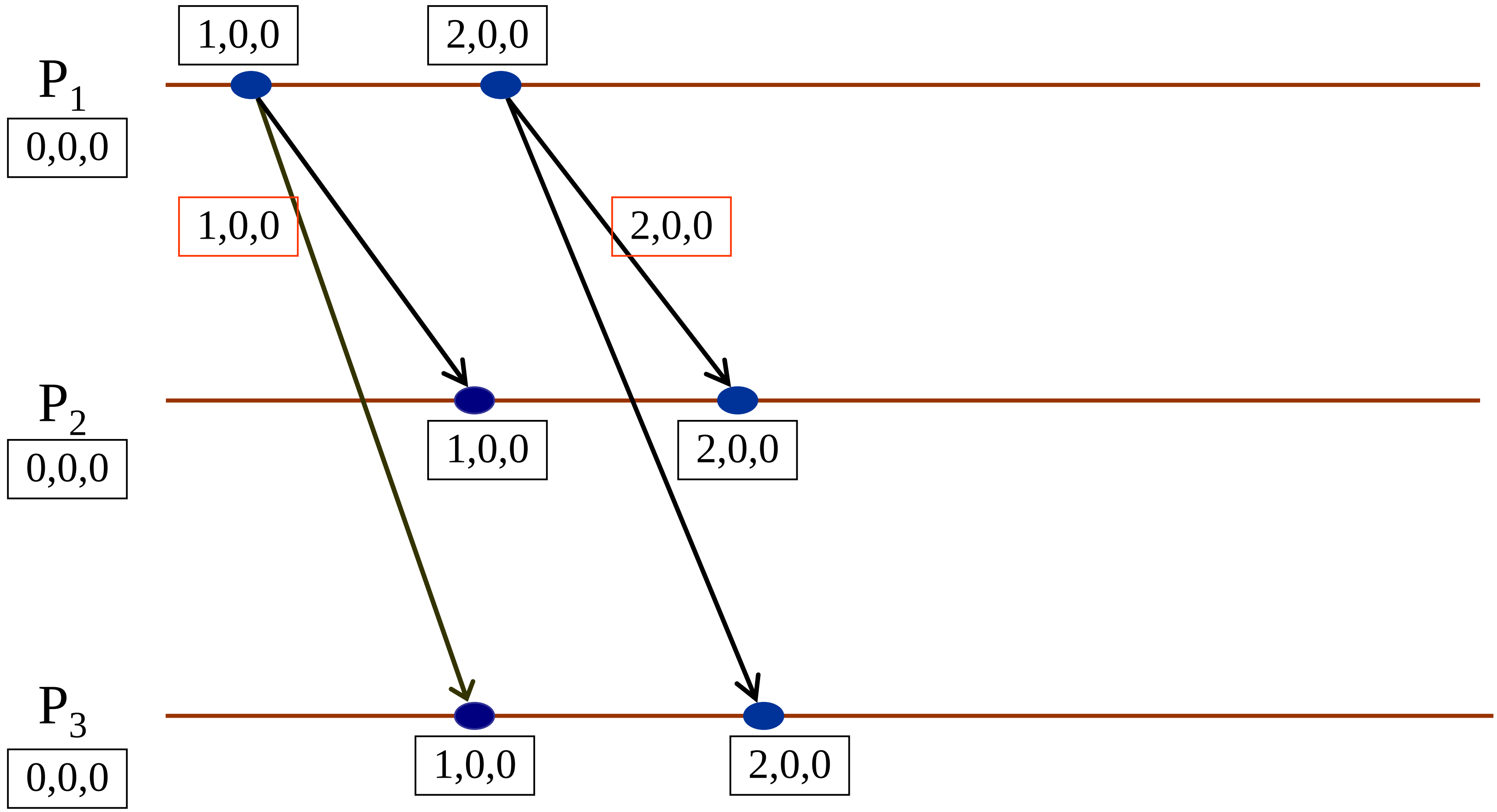
Applying Vector Clocks

to Broadcast System

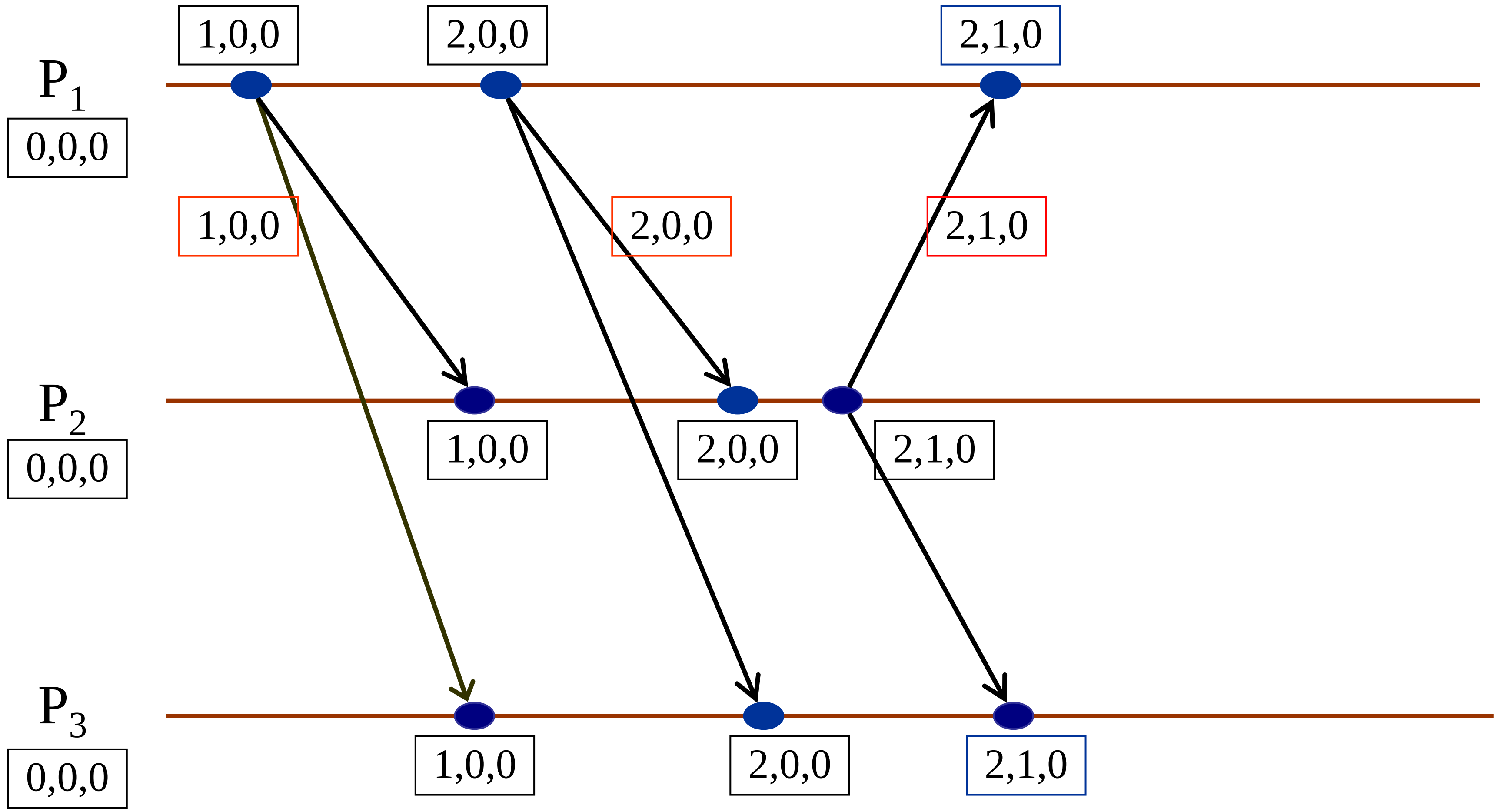


Applying Vector Clocks

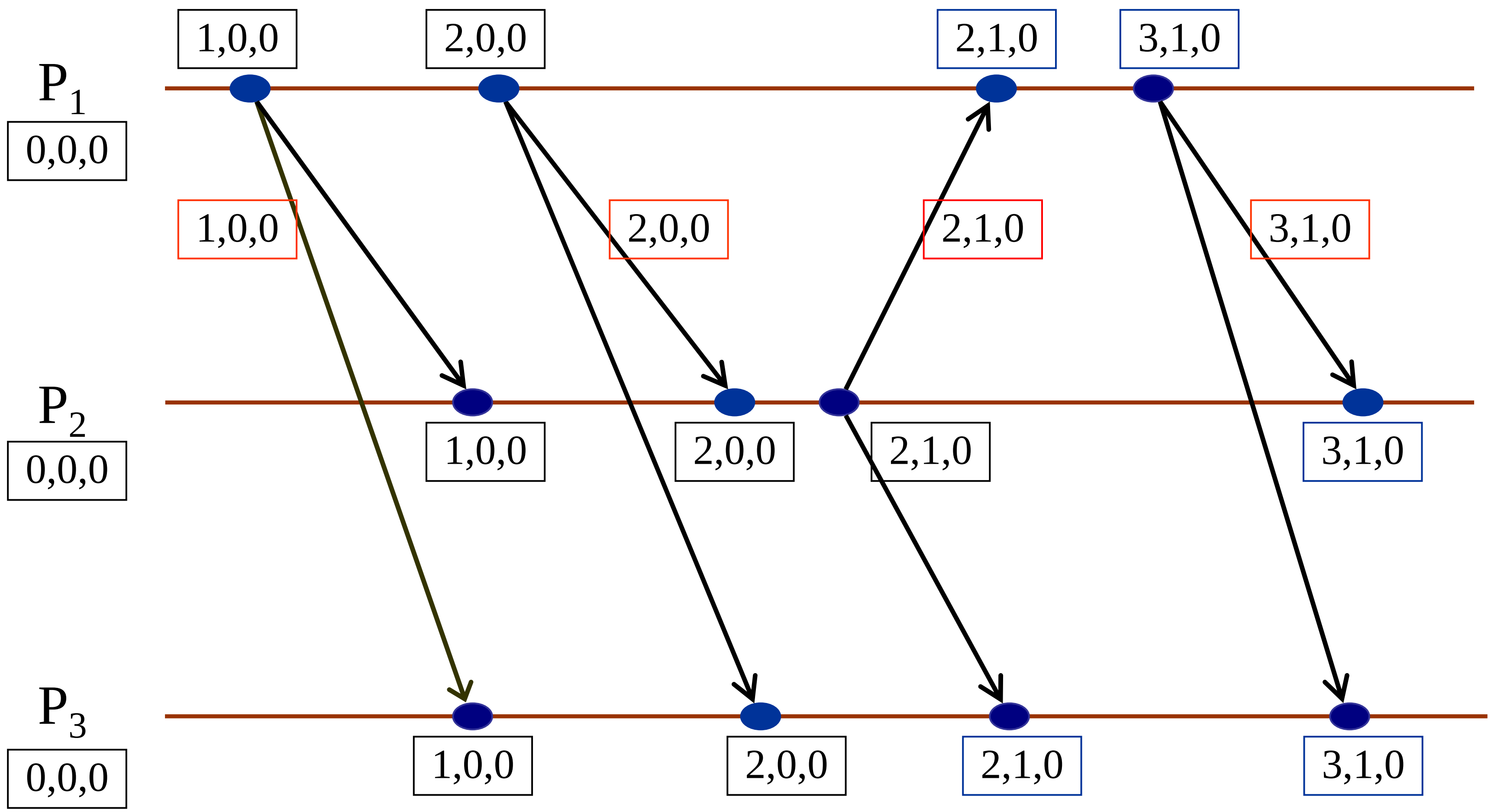
to Broadcast System



Applying Vector Clocks
to Broadcast System



Applying Vector Clocks
to Broadcast System



Some Observations

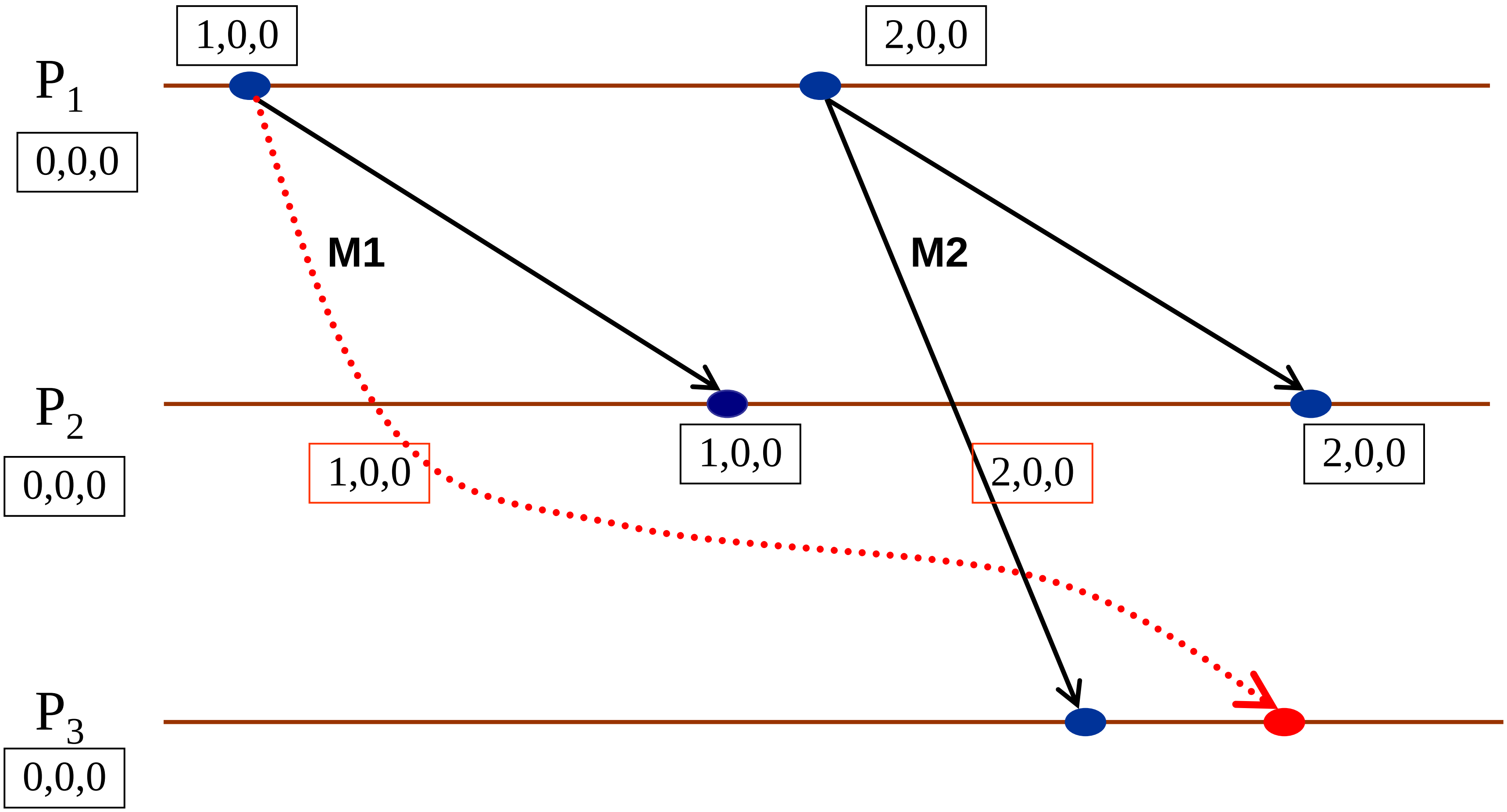
- VT_i [i] indicates number of messages sent by process Pi .

- VT_j [i] indicates number of messages received by process Pj sent by process Pi .

Applying Vector Clocks to
Ensure Globally Ordered Delivery of Messages

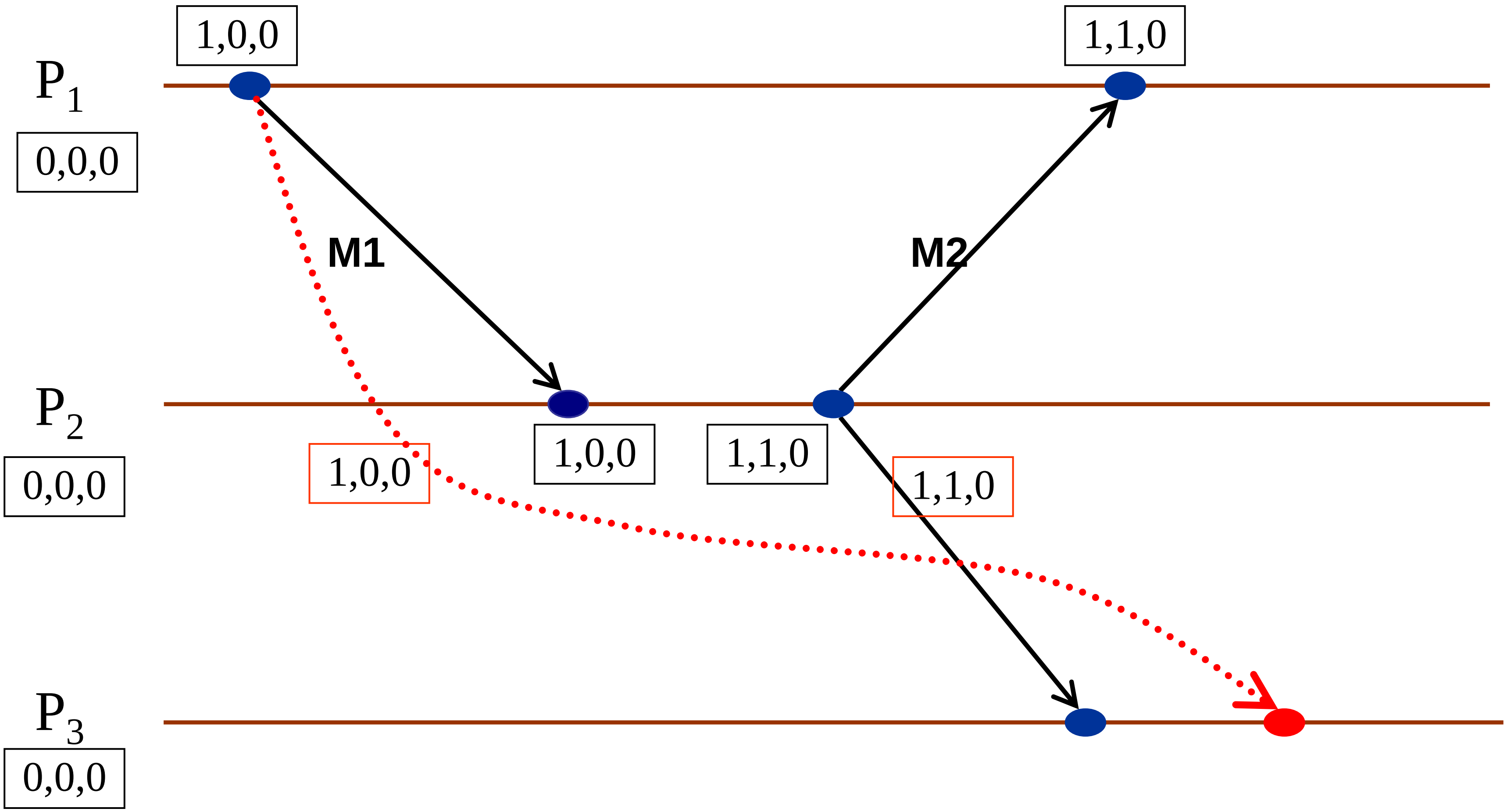
- Process Pi broadcasts a message M .
- A recipient process Pj delays the delivery of message M until following conditions are satisfied
  - ✓ $VT_j[i] = VT_M[i] - 1$
  - ✓ $VT_j[k] \geqslant VT_M[k], \forall k \in (1..N) \wedge (k \neq i)$

Applying Vector Clocks to
Ensure Globally Ordered Delivery of Messages



- Message M_2 arrives early at P_3.

Applying Vector Clocks to
Ensure Globally Ordered Delivery of Messages



- Message M_2 arrives early at P_3.

Applying Vector Clocks

Third Refinement

Introducing a new variables VTP and VTM

SETS

PROCESS ; MESSAGE

VARIABLES

sender, receive,
order, buffer,
VTP, VTM

INVARIANT

$VTP \in PROCESS \rightarrow (PROCESS \rightarrow \mathbb{N})$

$\wedge\ VTM \in MESSAGE \mathrel{+\!\!\!\rightarrow} (PROCESS \rightarrow \mathbb{N})$

Applying Vector Clocks

Third Refinement

Refinement of Operation Send

Send (pp,mm) ≙

SELECT mm ∉ dom(sender)

THEN

order := order ∪ ((sender~[{pp}] * {mm})
∪ (receive[{pp}] * {mm}))

|| sender := sender ∪ {mm ↦ pp}

END;

⇒

Send(pp,mm) ≙
SELECT mm ∉dom(sender)
∧ VTP(pp)(pp)>= 0
∧ VTP(pp)(pp)<MAXINT
THEN
LET nVTP
BE
nVTP = VTP(pp) ⩤ { pp ↦ VTP(pp)(pp)+1}
IN VTM(mm) := nVTP || VTP(pp) := nVTP

|| sender := sender ∪ {mm ↦ pp}
END;

Applying Vector Clocks

Third Refinement

Refinement of Operation Receive

$$
\begin{array}{l}
\textbf{Receive (pp,mm)} \triangleq \\
\textbf{SELECT } mm \in dom(sender) \wedge (pp \mapsto mm) \notin receive \wedge pp \neq sender(mm) \\
\qquad \wedge \forall m.(m \in MESSAGE \wedge (m \mapsto mm) \in order \wedge pp \neq sender(m) \Rightarrow (pp \mapsto m) \in receive) \\
\textbf{THEN } receive := receive \cup \{pp \mapsto mm\} \parallel buffer := buffer - \{pp \mapsto mm\} \\
\textbf{END}
\end{array}
$$

⬇

$$
\begin{array}{l}
\textbf{Recieve(pp,mm)} \triangleq \\
\textbf{SELECT} \\
\qquad mm \in dom(sender) \wedge (pp \mapsto mm) \notin receive \wedge pp \neq sender(mm) \\
\qquad \wedge (pp \mapsto mm) \in buffer \\
\qquad \wedge \forall p.(p \in PROCESS \wedge p \neq sender(mm) \Rightarrow VTP(pp)(p) \geqslant VTM(mm)(p)) \\
\qquad \wedge VTP(pp)(sender(mm)) = VTM(mm)(sender(mm)) - 1 \\
\textbf{THEN} \\
\qquad receive := receive \cup \{pp \mapsto mm\} \parallel buffer := buffer - \{pp \mapsto mm\} \\
\qquad \parallel VTP(pp) := VTP(pp) \mathbin{\lessdot} (\{q \mid q \in PROCESS \wedge VTP(pp)(q) < VTM(mm)(q)\} \lhd VTM(mm)) \\
\textbf{END}
\end{array}
$$

Applying Vector Clocks

Third Refinement

INVARIANT

$\forall$**m1,m2,p**$\cdot$**(m1**$\in$ **MESSAGE** $\wedge$ **m2**$\in$ **MESSAGE** $\wedge$ **p** $\in$ **PROCESS**
$\wedge$ **(m1** $\mapsto$ **m2)** $\in$ **order** $\Rightarrow$ **VTM (m1)(p)** $\leqslant$ **VTM(m2)(p))**

$\wedge$ $\forall$**p1,m,p**$\cdot$**(p1**$\in$ **PROCESS** $\wedge$ **p** $\in$ **PROCESS** $\wedge$ **m**$\in$ **MESSAGE** $\wedge$ **m**$\in$ **dom(sender)**
$\wedge$ **p1** $\neq$ **sender(m)** $\wedge$ **VTP(p1)(p)** $\geqslant$ **VTM(m)(p)** $\Rightarrow$ **(p1** $\mapsto$ **m)** $\in$ **receive)**

$\wedge$ $\forall$ **m,p** $\cdot$**(p** $\in$ **PROCESS** $\wedge$ **m**$\in$ **MESSAGE** $\wedge$ **m** $\in$ **dom(sender)**
$\Rightarrow$ **VTM(m)(p)** $\leqslant$ **VTP(p)(p))**

$\wedge$ $\forall$**p1,p2**$\cdot$**(p1** $\in$ **PROCESS** $\wedge$ **p2** $\in$ **PROCESS** $\wedge$ **p1**$\neq$**p2** $\Rightarrow$ **VTP(p1)(p2)** $\leqslant$ **VTP(p2)(p2))**

Conclusions

- We outlined how an abstract causal order is correctly implemented through vector clocks.

- Ordered delivery of messages may provide enough information needed at the time of recovery from failures.

- Adequacy of Event B to provide a complete framework for developing mathematical models of distributed algorithms.

- Illustration of use of Event B for rigorous description of problem, gradual refinement to more concrete specifications and verification of solution for correctness.