M.S. Ramaiah Institute of Technology
(Autonomous Institute, Affiliated to VTU)
Department of Computer Science and Engineering

Course Name: Distributed Systems
Course Code: CSE751/CSE20
Credits: 3:0:0/3:0:0:1

Term: Oct 2021– Feb 2022

Faculty:
Sini Anna Alex

2.1 MESSAGE ORDERING PARADIGMS

Notations

We model the distributed system as a graph (N, L). The following notation is used to refer to messages and events:

- When referring to a message without regard for the identity of the sender and receiver processes, we use m^i. For message m^i, its send and receive events are denoted as s^i and r^i, respectively.
- More generally, send and receive events are denoted simply as s and r. When the relationship between the message and its send and receive events is to be stressed, we also use M, send(M) , and receive(M) respectively.

For any two events a and b, where each can be either a send event or a receive event, the notation $a \sim b$ denotes that a and b occur at the same process, i.e., $a \in E_i$ and $b \in E_i$ for some process i.

The send and receive event pair for a message is said to be **a pair of *corresponding* events**. The send event corresponds to the receive event, and vice-versa. For a given execution E, let the set of all send–receive event pairs be denoted as $T = \{(s,r) \in E_i \times E_j \mid s \text{ corresponds to } r\}$.

Message ordering paradigms

The order of delivery of messages in a distributed system is an important aspect of system executions because it determines the messaging behavior that can be expected by the distributed program.

Several orderings on messages have been defined:

(i) non-FIFO,

(ii) FIFO,

(iii) causal order, and

(iv) synchronous order.

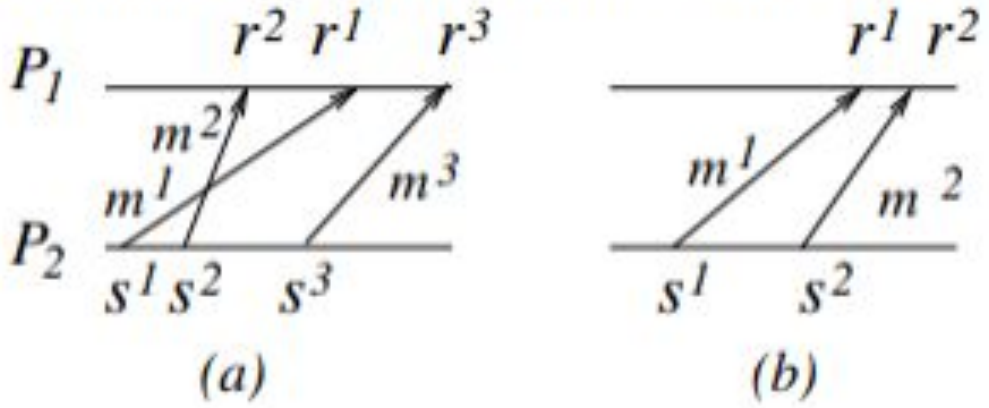
RAMAIAH
Institute of Technology

2.1.1 Asynchronous and FIFO Executions

Definition (A-execution) : An asynchronous execution (or A-execution) is an execution $E_{\prec}$ for which the causality relation is a partial order.

- On any logical link between two nodes in the system, messages may be delivered in any order, *not necessarily* first-in first-out. Such executions are also known as *non-FIFO executions*. , e.g., network layer IPv4 connectionless service
- **All physical links obey FIFO**

(a) A-execution that is not FIFO (b) A-execution that is FIFO



2.1.3 Causal order (CO)

A CO execution is an a execution in which, for all (s,r) and (s' ,r') $\in$ T , (r $\sim$ r' and s $\prec$ s') $\Longrightarrow$ r $\prec$ r'

- **If send events s and s' are related by causality ordering (not physical time ordering), their corresponding receive events r and r' occur in the same order at all common destinations.**
- **If s and s' are not related by causality, then CO is vacuously satisfied.**

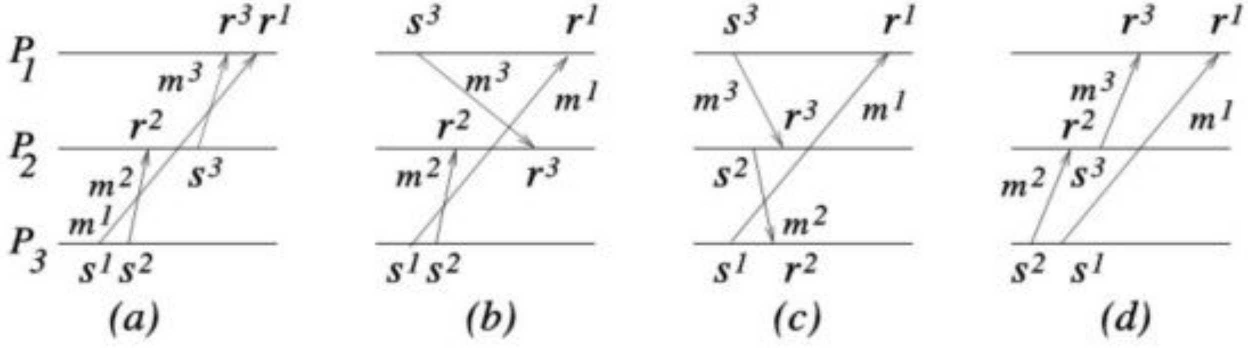


Fig (6.2) (a) Violates CO as $s^1 \prec s^3$; $r^3 \prec r^1$ (b) Satisfies CO. (c) Satisfies CO. No send events related by causality. (d) Satisfies CO.

Examples

- Figure (a) shows an execution that violates CO because $s^1 < s^3$ and at the common destination P_1, we have $r^3 < r^1$.
- Figure (b) shows an execution that satisfies CO. Only s^1 and s^2 are related by causality but the destinations of the corresponding messages are different.
- Figure (c) shows an execution that satisfies CO. No send events are related by causality.
- Figure (d) shows an execution that satisfies CO. s^2 and s^1 are related by causality but the destinations of the corresponding messages are different. Similarly for s^2 and s^3.

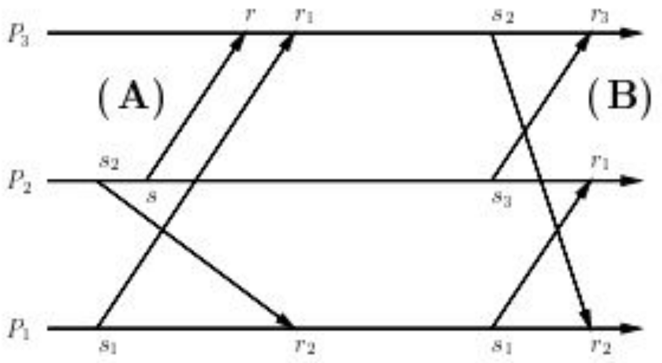
Crowns in Distributed Computation

- A computation is synchronous iff there does not exist a sequence of send and corresponding receive events such that

$$s_0 \longrightarrow_{\mathcal{E}} r_1, s_1 \longrightarrow_{\mathcal{E}} r_2, \ldots, s_{k-2} \longrightarrow_{\mathcal{E}} r_{k-1}, s_{k-1} \longrightarrow_{\mathcal{E}} r_0.$$

  – such a structure is called crown.

- Example:



$s_1 \rightarrow r_2, s_2 \rightarrow r_1$ $s_1 \prec r_2, s_2 \prec r_3, s_3 \prec r_1$

(A) : Crown of size 2
(B) : Strong Crown of size 3

Crown: Definition

Let E be an execution. A crown of size k in E is a sequence $\langle s^i, r^i \rangle$, $i \in \{0, \ldots, k-1\}$ of pairs of corresponding send and receive events such that: $s^0 \prec r^1$, $s^1 \prec r^2$, $\ldots$, $s^{k-2} \prec r^{k-1}$, $s^{k-1} \prec r^0$.

- In a crown, s^i and r^{i+1} may or may not be on same process
- Non-CO execution must have a crown
- CO executions (that are not synchronous) have a crown (see Fig 6.2(b))
- Cyclic dependencies of crown $\Rightarrow$ cannot schedule messages serially $\Rightarrow$ not RSC

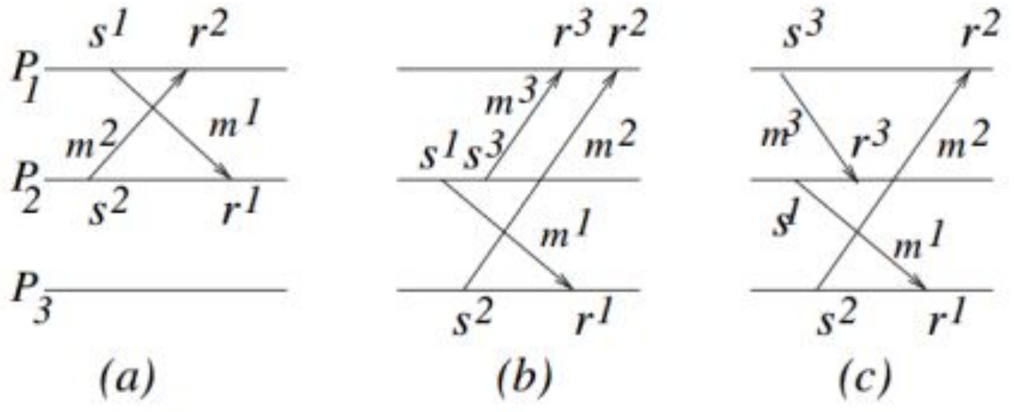


Figure 6.5: Illustration of non-RSC A-executions and crowns. .

(a) Crown of size 2.

(b) Another crown of size 2.

(c) Crown of size 3.

2.3.2 Algorithm for binary rendezvous

These algorithms typically share the following features

- At each process, there is a set of tokens representing the current interactions that are enabled locally.
- If multiple interactions are enabled, a process chooses one of them and tries to “synchronize” with the partner process.

The problem reduces to one of scheduling messages satisfying the following constraints:

- Schedule on-line, atomically, and in a distributed manner, i.e., the schedul-ing code at any process does not know the application code of other processes.
- Schedule in a deadlock-free manner (i.e., crown-free), such that both the sender and receiver are enabled for a message when it is scheduled.
- Schedule to satisfy the progress property (i.e., find a schedule within a bounded number of steps) in addition to the safety (i.e., correctness) property.
- Additional features of a good algorithm are: (i) symmetry or some form of fairness, i.e., not favoring particular processes over others during scheduling, and (ii) efficiency, i.e., using as few messages as possible, and involving as low a time overhead as possible.

We now outline a simple algorithm by Bagrodia that makes the following **assumptions:**

1. **Receive commands are forever enabled from all processes.**

2. **A send command, once enabled, remains enabled until it completes, i.e., it is not possible that a send command gets disabled (by its guard getting falsified) before the send is executed.**

3. **To prevent deadlock, process identifiers are used to introduce asymmetry to break potential crowns that arise.**

4. **Each process attempts to schedule only one send event at any time.**

The algorithm illustrates how crown-free message scheduling is achieved on-line.

<u>The message types used are: (i) M, (ii) *ack*(M), (iii) *request*(M), and (iv) *permission*(M).</u> A process blocks when it knows that it can successfully synchronize the current message with the partner process. Each process maintains a queue that is processed in FIFO order only when the process is unblocked. When a process is blocked waiting for a particular message that it is currently synchronizing, any other message that arrives is queued up.

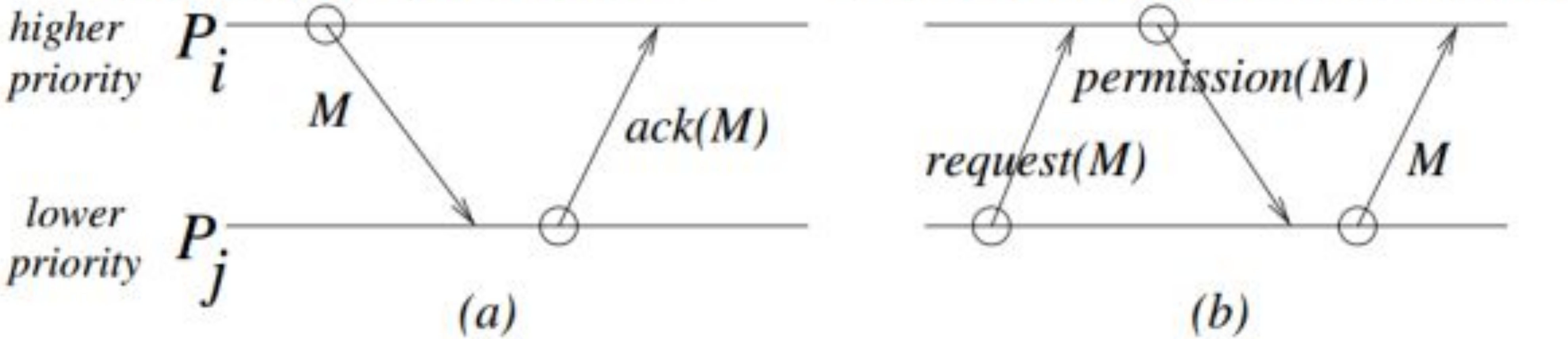
Execution events in the synchronous execution are only the *send* of the message M and *receive* of the message M. The send and receive events for the other message types – *ack*(M), *request*(M), and *permission*(M) which are con-trol messages – are under the covers, and are not included in the synchronous execution. The messages *request*(M), *ack*(M), and *permission*(M) use M's unique tag; the message M is not included in these messages. We use cap-ital SEND(M) and RECEIVE(M) to denote the primitives in the application execution, the lower case send and receive are used for the control messages.

To send to a lower priority process, messages M and ack(M) are involved in that order. The sender issues send(M) and blocks until ack(M) arrives. Thus, when sending to a lower priority process, the sender blocks waiting for the partner process to synchronize and send an acknowledgement.

To send to a higher priority process, messages request(M), permission(M), and M are involved, in that order. The sender issues send(request(M)), does not block, and awaits permission. When permission(M) arrives, the sender issues send(M).

Rules to prevent message cyles.

(a) High priority process blocks. **(b) Low priority process does not block.**



Bagrodia's Algorithm for Binary Rendezvous: Code

RAMA Institute of

(message types)
M, $ack(M)$, $request(M)$, $permission(M)$

1. **P_i wants to execute SEND(M) to a lower priority process P_j:**
   P_i executes $send(M)$ and blocks until it receives $ack(M)$ from P_j. The send event SEND(M) now completes.
   Any M' message (from a higher priority processes) and $request(M')$ request for synchronization (from a lower priority processes) received during the blocking period are queued.

2. **P_i wants to execute SEND(M) to a higher priority process P_j:**
   1. P_i seeks permission from P_j by executing $send(request(M))$.
      // to avoid deadlock in which cyclically blocked processes queue messages.
   2. While P_i is waiting for permission, it remains unblocked.
      1. If a message M' arrives from a higher priority process P_k, P_i accepts M' by scheduling a RECEIVE(M') event and then executes $send(ack(M'))$ to P_k.
      2. If a $request(M')$ arrives from a lower priority process P_k, P_i executes $send(permission(M'))$ to P_k and blocks waiting for the message M'. When M' arrives, the RECEIVE(M') event is executed.
   3. When the $permission(M)$ arrives, P_i knows partner P_j is synchronized and P_i executes $send(M)$. The SEND(M) now completes.

3. **Request(M) arrival at P_i from a lower priority process P_j:**
   At the time a $request(M)$ is processed by P_i, process P_i executes $send(permission(M))$ to P_j and blocks waiting for the message M. When M arrives, the RECEIVE(M) event is executed and the process unblocks.

4. **Message M arrival at P_i from a higher priority process P_j:**
   At the time a message M is processed by P_i, process P_i executes RECEIVE(M) (which is assumed to be always enabled) and then $send(ack(M))$ to P_j.

5. **Processing when P_i is unblocked:**
   When P_i is unblocked, it dequeues the next (if any) message from the queue and processes it as a message arrival (as per Rules 3 or 4).

Thank you