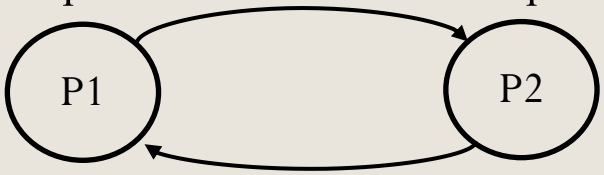
# Operating Systems
# Message Passing

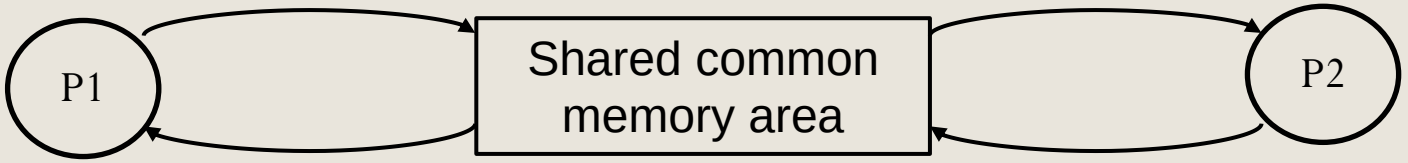
Shyan-Ming Yuan
CS Department, NCTU
smyuan@gmail.com
Office hours: Wed. 8~10 (EC442)

# Inter Process Communication (IPC)

- Information sharing among two or more processes requires interprocess communication (IPC).
- There are two basic IPC mechanisms:
  - Message Passing Approach:
    - The shared information is physically copied from the sender process's address space to the address spaces of all the receiver processes.




  - Shared Memory Approach:
    - The shared information is placed in a common memory area that is accessible to all the processes.



# The Message Passing Model

- Messages are collections of data objects and their structures
- A message consists of
  - a fixed-length header containing control information and
  - a fixed-length or variable-length message body containing a collection of typed data objects.
- A message of fixed-length is simple to manage.
  - A pool of fixed-sized buffers can be used in such a way that the processing and storage overhead is minimized.
- A message of variable length is difficult to manage.
  - Dynamic memory allocation (a heap with garbage collection) mechanism is needed.
    - It needs more overheads and creates memory fragmentations.

# Direct Communication Model

- Direct Communication model uses process names (global process IDs) to specify communicating participants.
  - In Symmetric Addressing, both the sender and receiver have to explicitly name their counterparts in the primitives.
    - send (in destination, in message)
    - receive (in source, **out** message)
  - In Asymmetric Addressing, only the sender needs to indicate the recipient.
    - send (in destination, in message)
    - receive (**out** source, **out** message)
- Disadvantages :
  - Changing the name of a process requires all related processes to update their programs.
  - Process names need to be known in advance.

# Naming / Addressing

- There are several common ways to identify a process.
  - "Hostname + Local-ID "
  - " Hostname (original) + Local-ID + Hostname (current) "
    - Initially, the 2nd hostname and the 1st hostname are both the hostname of the original creator.
      - When a process migrates to a new host, the 2nd hostname is changed to the hostname of the new host.
      - Every host maintains a mapping table for migration information.
    - A message is always sent to 2nd host with "1st hostname + local-ID".
      - Upon receiving a message, if the destination process was moved, the new 2nd hostname is used to forward the message.
      - The new global process ID can be sent back to the sender on ACK.
  - A global name server is used for mapping a process's human readable string name to its current global ID (i.e. " Hostname + Local-ID " ).

# The Indirect Communication Model

- Indirect Communication model does not send a message from a sender to its receiver directly.
  - A shared data structure is used among senders and receivers.
  - Messages are sent to the shared data structure by senders and retrieved from the shared data structure by receivers.
  - Mailbox, port, message queues are the commonly used shared data structures.
- Indirect Communication primitives are:
  - Create/open/close/destroy (in mailbox)
  - Send (in mailbox, in message)
  - Receive (in mailbox, **out** message)
- Senders and receivers are decoupled.

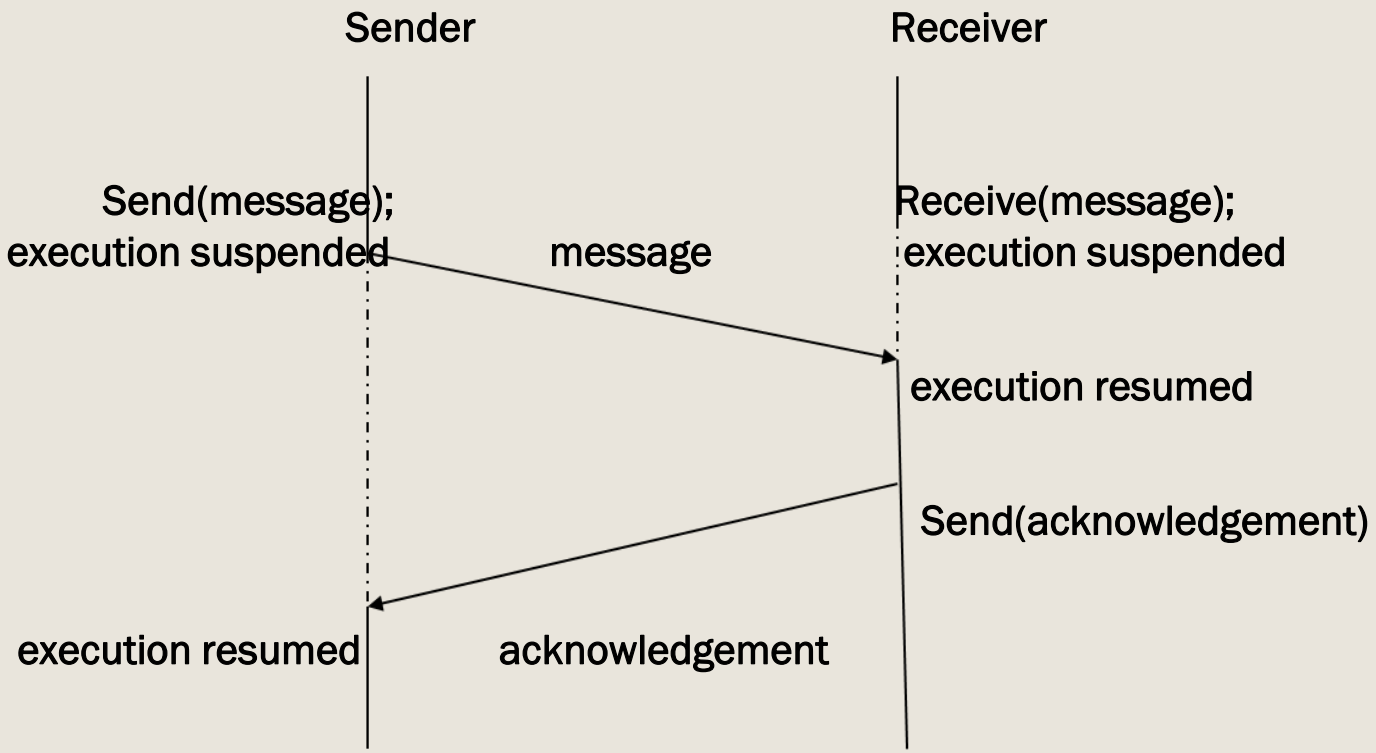
# Persistent and Transient Messaging

- In persistent messaging, messages are stored as long as necessary by the communication system (e.g., e-mail) until they can be delivered.
  - A sender can terminate after executing the send primitive.
  - A receiver may retrieve a message whenever it needs.
- In transient messaging, messages exist only while the sender and receiver are running.
  - Transient messages are discarded when they cannot be delivered.
    - Communication errors or inactive receivers can cause messages to be discarded.
  - UDP and TCP are both transient.
- In general, indirect communication systems are more likely to support persistent messaging.

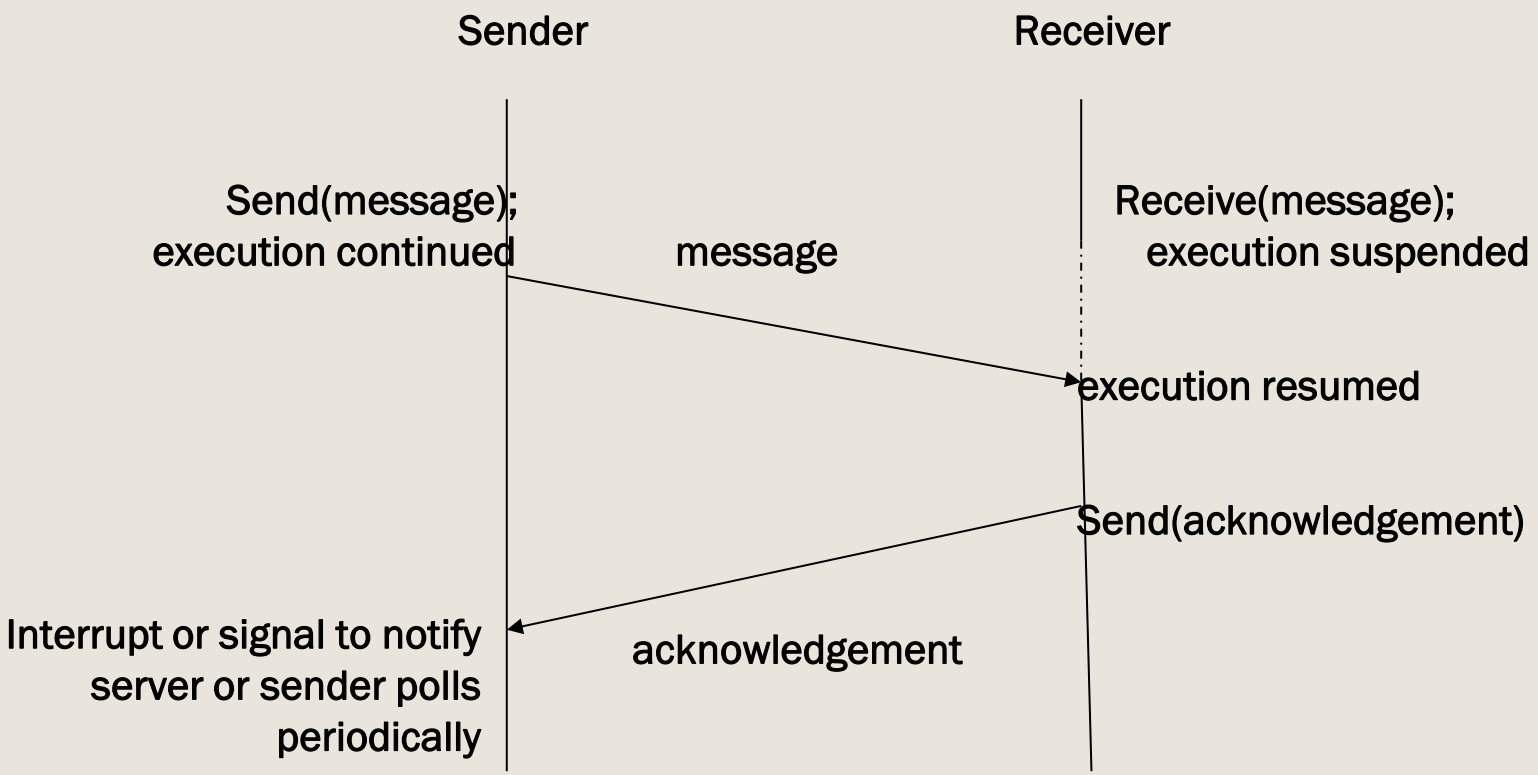
# Blocking vs Nonblocking Primitives

- In blocking send model, a sender is blocked until
  - the communication system **notifies acceptance** of the message, or
  - the message has been **delivered** to the receiver, or
  - the receiver processes it and returns a **response**, or **timed out**.
- In blocking receive model, a receiver is blocked until
  - the desired message is **received** or **timed out**.
- In non-blocking send, a sender resumes execution as soon as the message is passed to the communication system.
- In non-blocking receive, a receiver resumes execution immediately.
  - A message is either **returned** or
  - an **error flag** is set to indicate that no message was returned.

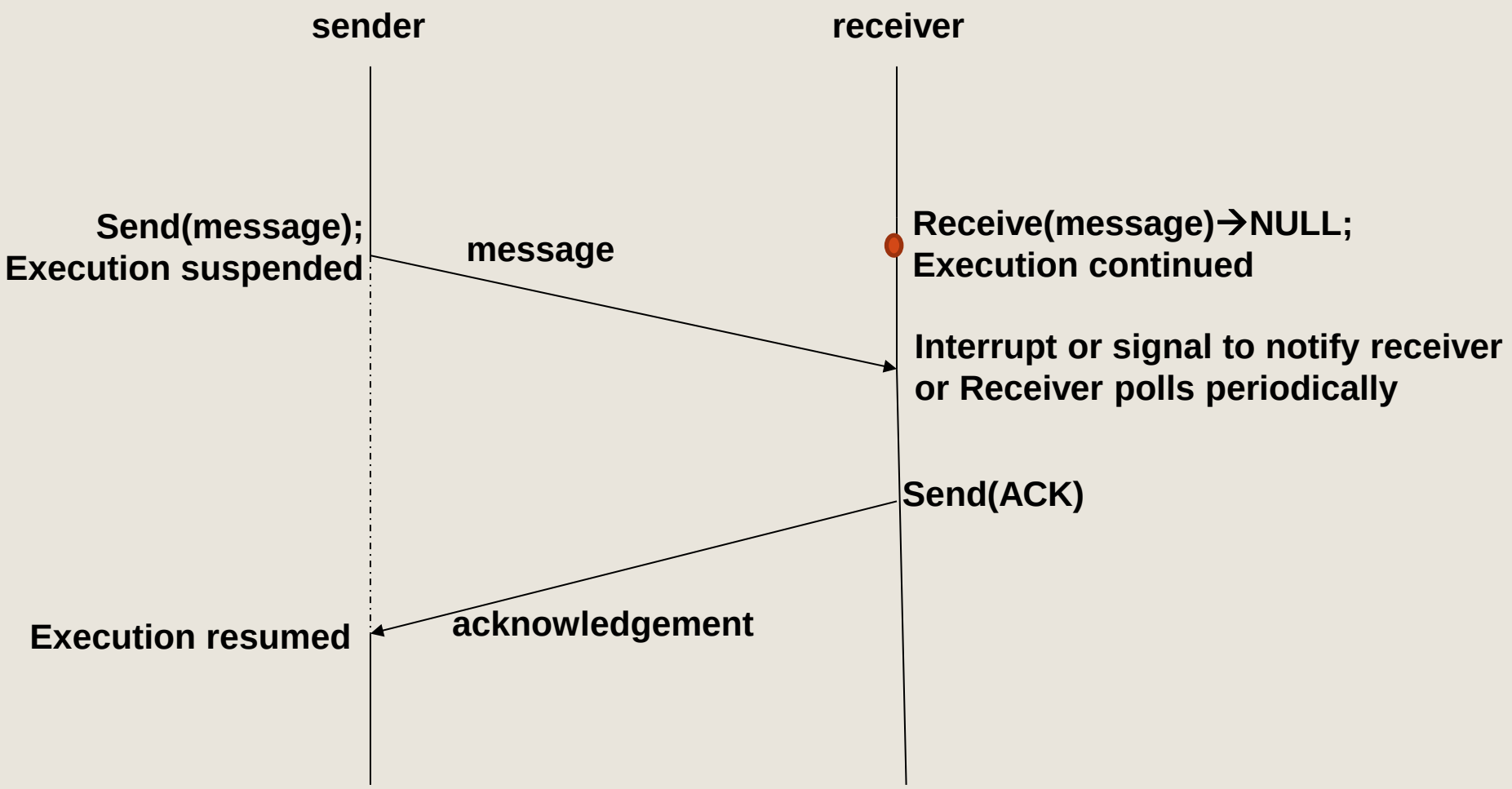
# Blocking Send + Blocking Receive



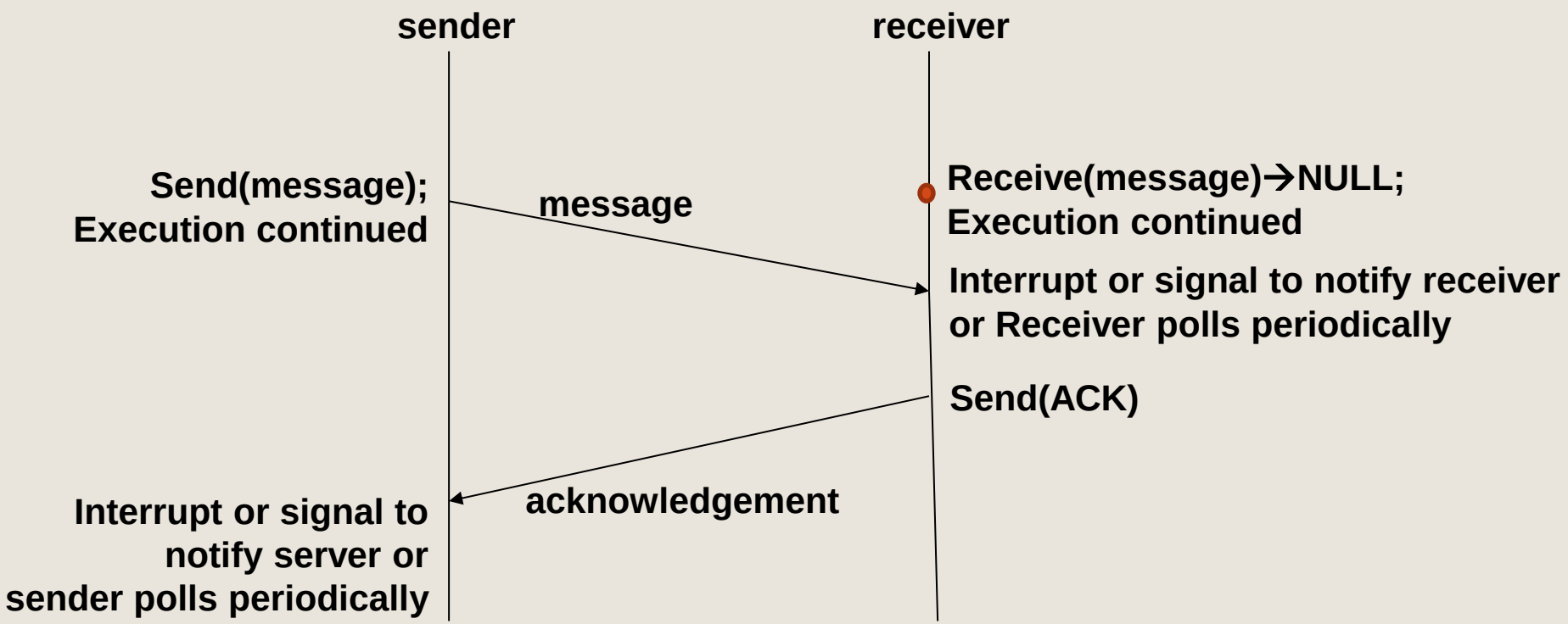
# Non-Blocking Send + Blocking Receive



# Blocking Send + Non-blocking Receive



# Non-Blocking + Non-Blocking Receive



# Pros and Cons of Blocking Primitives

- Pros:
  - The blocking primitives are much easy to implement.
- Cons:
  - The sender and/or the receiver can block forever if the other party crashes or messages are lost.
  - The concurrency degree between sender and receiver is limited.
  - The communication deadlock is more likely to happen.

```
P1:
   send() to P2;
   receive() from P2;
```

```
P2:
   send() to P1;
   receive() from P1;
```

# Pros and Cons of Non-blocking OPs

- Pros:
  - The sender and/or the receiver can continue their executions without waiting for each other.
  - The system is less likely to run into a deadlock.
- Cons:
  - The sender does not know whether or not the receiver has actually received the message.
  - The sender cannot modify the message buffer until the message has been sent.
    - Additional overhead is required to notify the sender and the receiver.
    - Additional data copy is required to reuse message buffer.

# Message Buffering

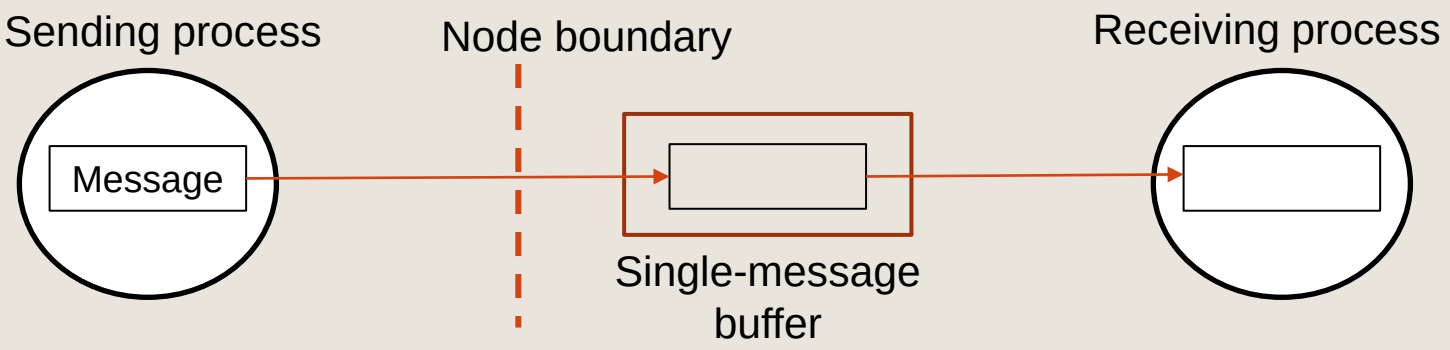
- In general, a message is transmitted from a sender to a receiver by copying the message from the address space of the sending process to the address space of the receiving process.
  - If a receiver is not ready to accept a message, the communication system may need to save the message for later usage.
- The message buffering strategy is strongly affected by whether blocking or non-blocking primitives are used.
  - In blocking send, no buffer is needed in the sender side.
  - In non-blocking send, some buffers are needed in the sender side.
  - In blocking receive, there may or may not have buffers in the receiver sider.
  - In non-blocking receive, some buffers are needed in the receiver side.

# Blocking Send + Blocking Receive (1)

- No Buffering Strategies:
  - Send-After-Receive policy: a message is only copied once.
    - The message remains in the sending process and the execution of send is delayed until the receiver executes the corresponding receive.
    - On receiving an ACK, the message is copied into the receiving process.
  - Send-Discard-Retry policy: a message may be copied several times.
    - After executing send, the sender waits for an ACK.
    - If the receiver is not ready, the message is simply discarded and a time out mechanism is used to resend message after a time out period.

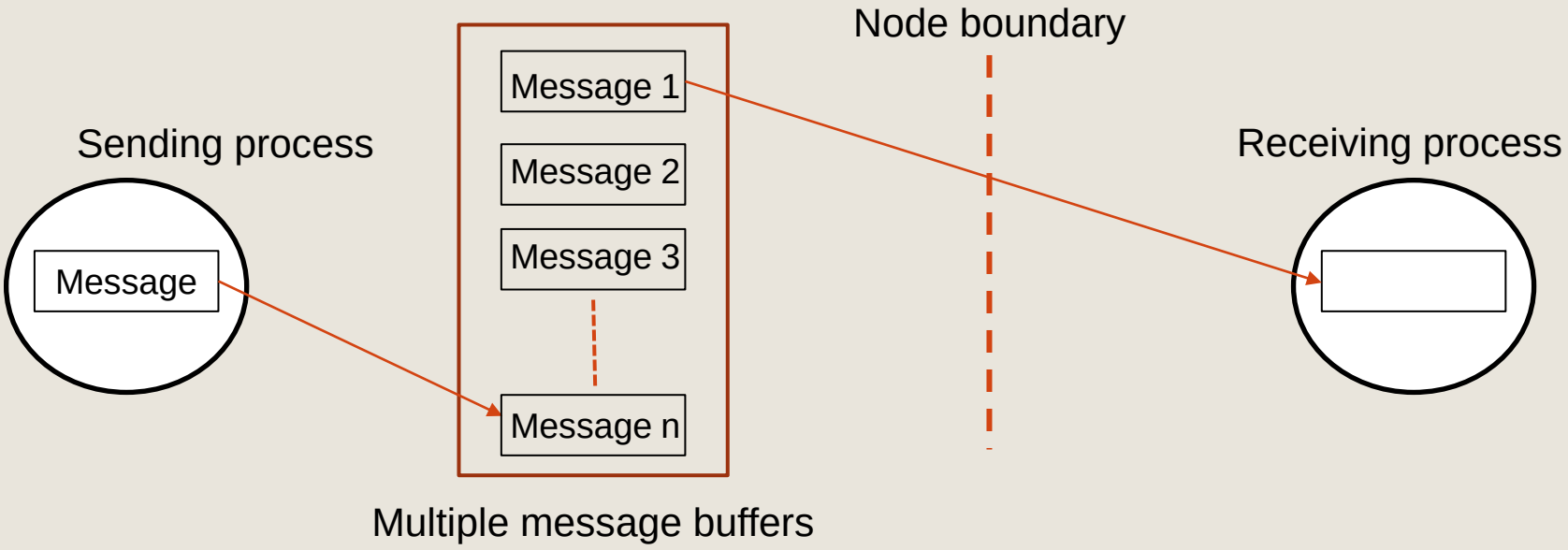
# Blocking Send + Blocking Receive (2)

- Single Buffer Strategy: a message is always copied twice.
  - A buffer is used on the receiver side to store one outstanding incoming message.
  - The sending process will be blocked until the buffer is available.



# Non-Blocking Send + Blocking Receive

- Multiple Buffer Strategy: a message is always copied twice.
  - A finite number of buffers are used in the sender side.
  - A single buffer may be used in the receiver side to reduce the control messages. An extra message copy is needed.



# Non-Blocking Receive

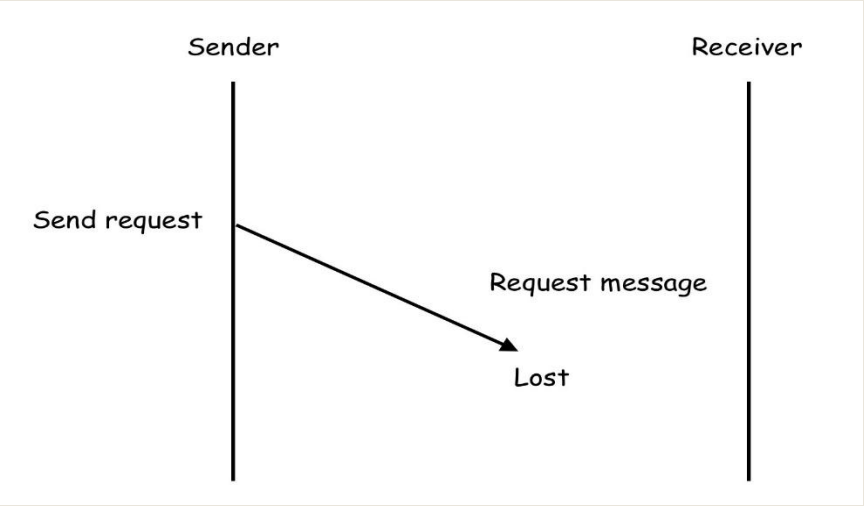
- At least one buffer is needed in the receiver side to make non-blocking receive possible.
  - Otherwise, the receive function must send a control message to the sender and wait for the ACK before it can determine to return either a message or a NULL.
    - This violates the definition of non-blocking.
- In general, a finite number of buffers are used in the receiver side to support non-blocking receive.
  - If blocking send primitive is preferred, then no buffer is needed in the sender side.
  - If non-blocking send is preferred, then a finite number of buffers are needed in the sender side also.

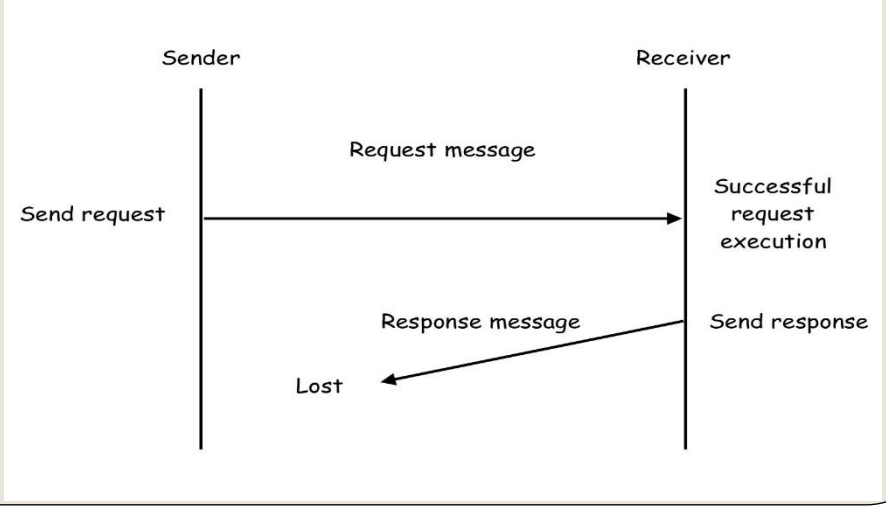
# The Client Server Computing Model

- The client-server model assigns asymmetric roles to communicating processes.
  - One process, the server, plays the role of a service provider which waits passively for the arrival of request messages.
  - The other processes, the clients, issue specific requests to the server and awaits its response.
- Many well known internet services are client server applications.
  - Telnet, HTTP, FTP, DNS, finger, gopher, etc.
- Most distributed systems use the client server model as their underlining communication model.

# Failure Handling in Client Server Model

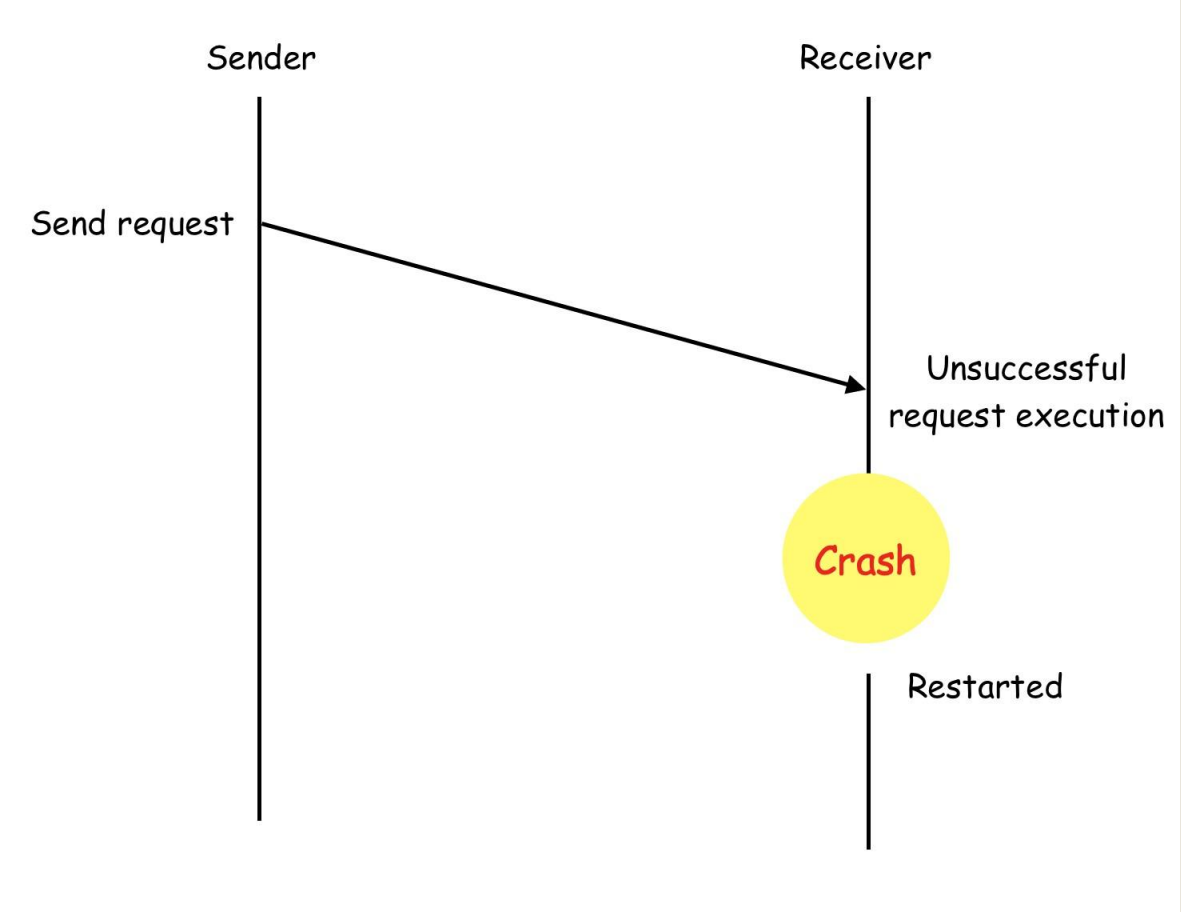
- There are three basic exceptions may happen in message exchanges of client server model.
  - LostReq: Lost of request (Req) message
  - LostResp: Lost of response message (Resp)
  - SerCrash: Unsuccessful execution of the request
- To handle these exceptions, a reliable message passing system is needed.



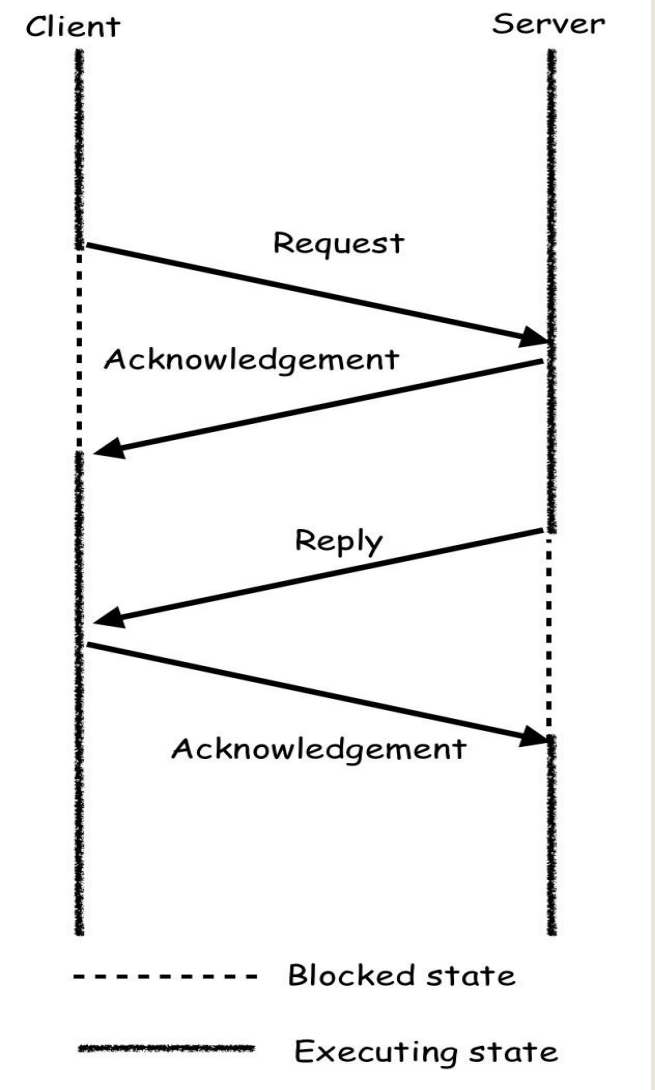
# Unsuccessful request execution

- The server crash after receiving the request message and before generate the response.



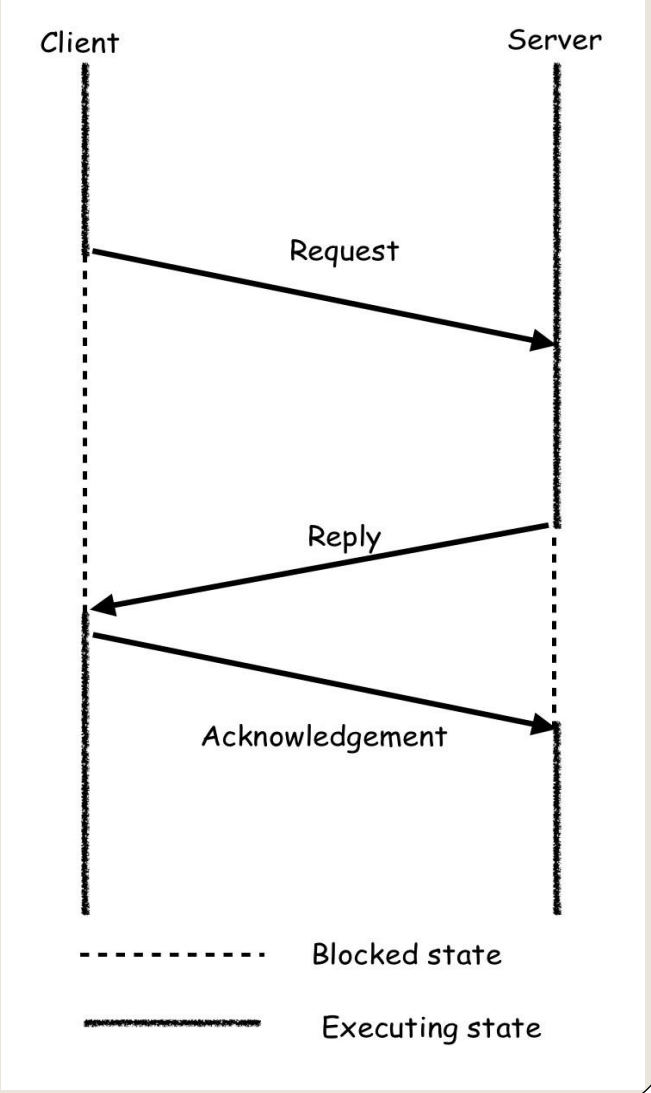
# The Req-Ack-Resp-Ack (RARA) Protocol

- Acknowledgement, timeout, and retransmission are most commonly used mechanisms in reliable message passing systems.
- The RARA protocol can be used to handle LostReq and LostResp failures.
  - The client uses a time out for resending the Req if LostReq occurs.
  - The server uses a timeout for resending the Resp if LostResp occurs.



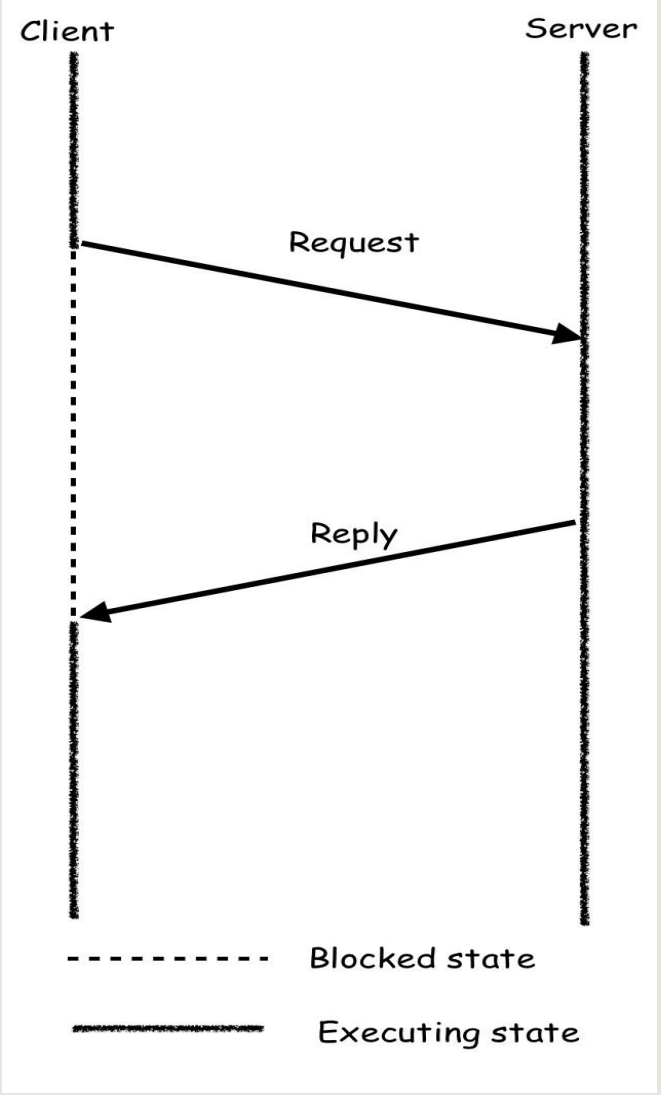
# The Req-Resp-Ack (RRA) Protocol

- Since the Resp is sufficient to acknowledge the reception of the Req, it is possible to eliminate the 1st Ack message.
- The RRA protocol can handle both LostReq and LostResp exceptions if the execution of the Req does not take an extremely long time.
  - To handle long processing time situation, the server can set a timer for sending an optional Ack to inform the client the reception of the request.
- Both client and server again use timeouts for resending Req or Resp.



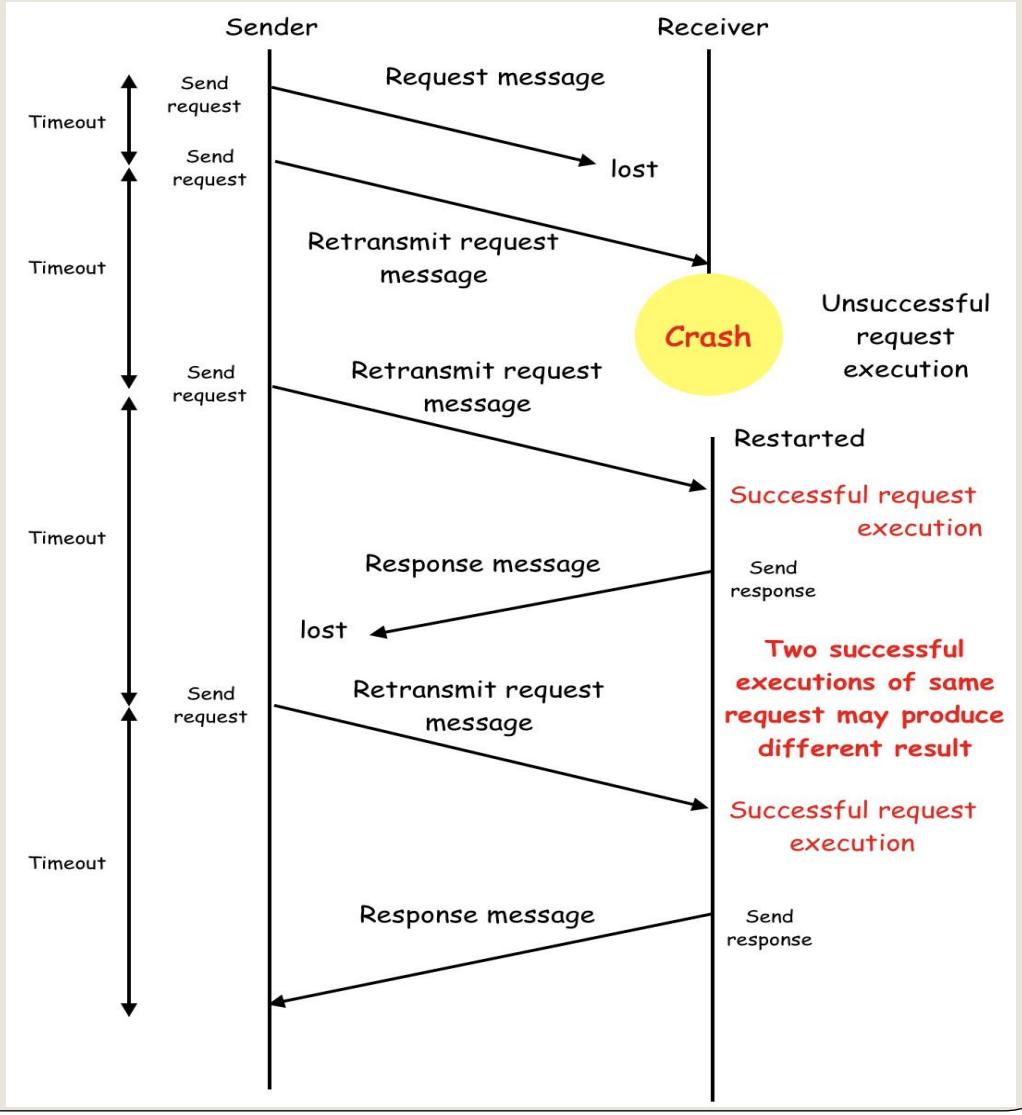
# The Req-Resp (RR) Protocol

- The Ack from client to server in the RRA protocol is convenient but not a necessity also.
  - If the Resp is lost, the client will resend the Req after timeout. The server can (process the Req once again and) return the Resp to the client.
- Both client and server again use timeouts for resending Req or Resp.



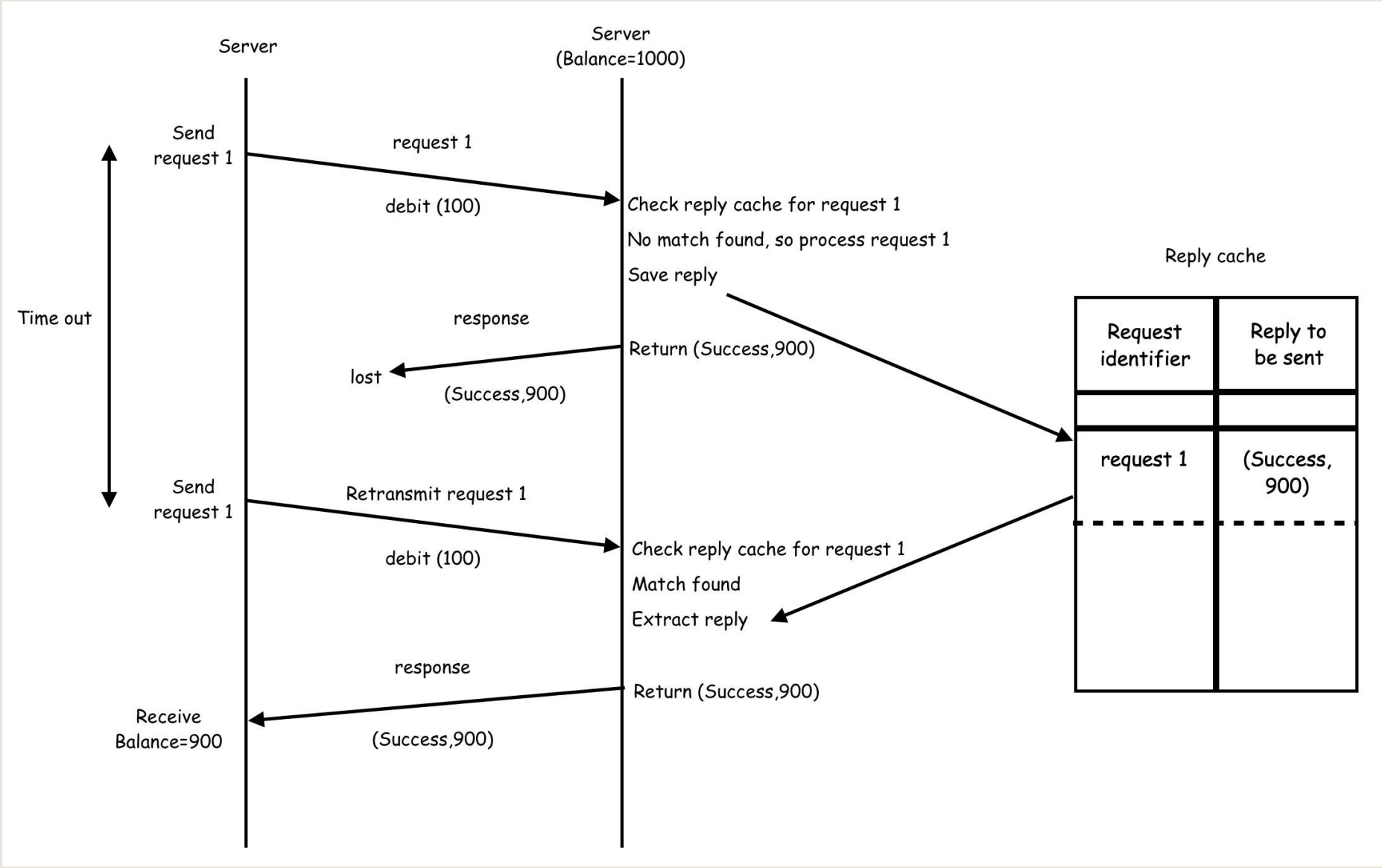
# The At-Least-Once Semantics

- The RR protocol ensures that the Req is executed at least once by the server.
- Since the results of the last execution of the Req is used by the client, it is also called last-one semantics.



# The Idempotent Service

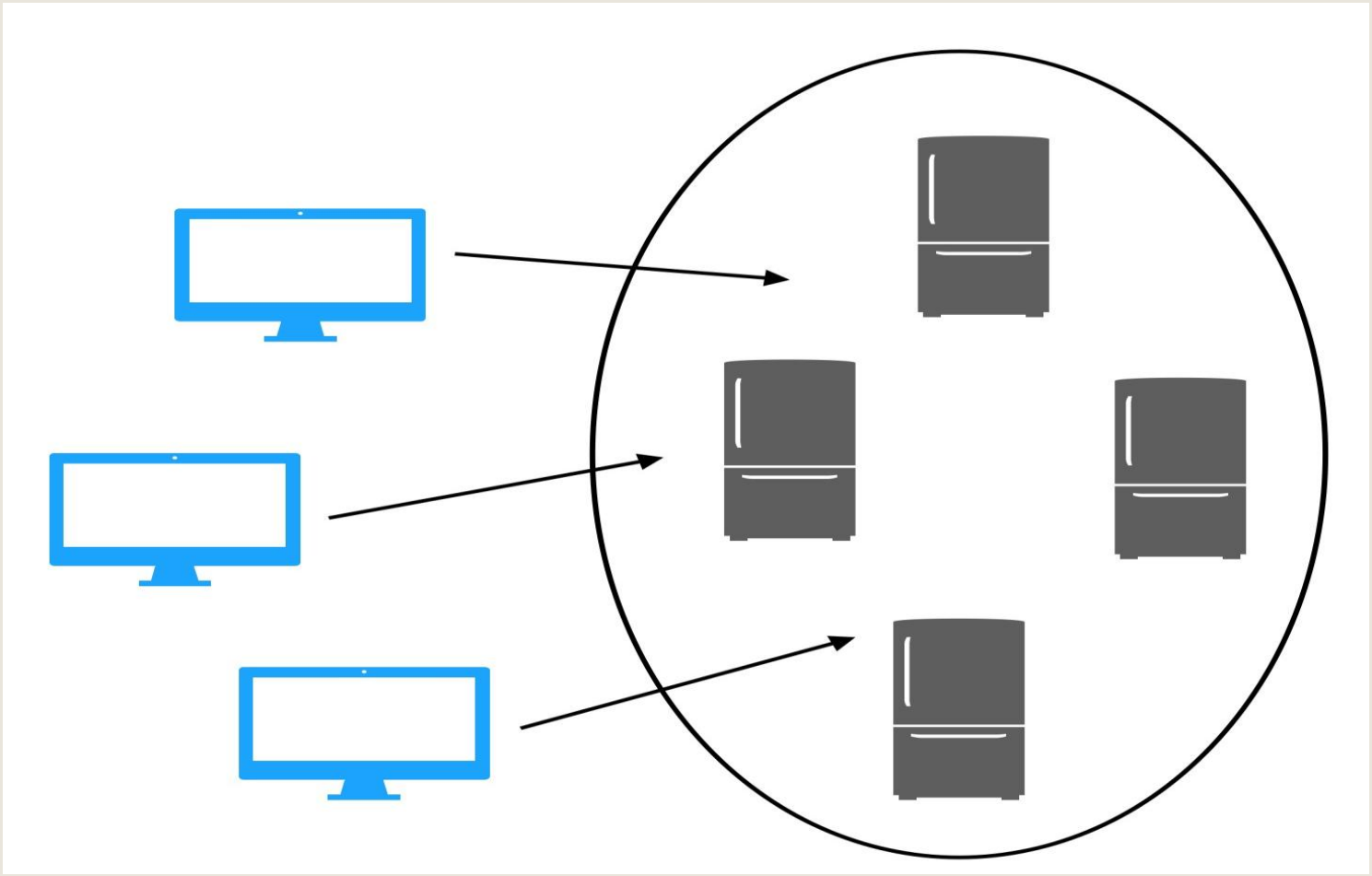
- The Idempotency means repeatability.
  - An idempotent operation produces the same results without any side effects no matter how many times it is performed with the same arguments.
  - Thus, the At-Least-Once semantics is proper for idempotent services.
- However, for non-idempotent services, the Exactly-Once semantics is needed.
- One possible implementation of Exactly-Once is as follows.
  - The client assigns a unique ID to every Req.
    - The server then can discard duplicated Req.
  - The server sets up a reply cache for each Resp.
    - The server then can avoid to execute the same Req twice.


Server
Server
(Balance=1000)
Send
request 1
request 1
debit (100)
Check reply cache for request 1
No match found, so process request 1
Save reply
Reply cache
Request identifier
Reply to be sent
Time out
response
Return (Success,900)
lost
(Success,900)
request 1
(Success, 900)
Send
request 1
Retransmit request 1
debit (100)
Check reply cache for request 1
Match found
Extract reply
response
Return (Success,900)
Receive
Balance=900
(Success,900)

# Group Communication

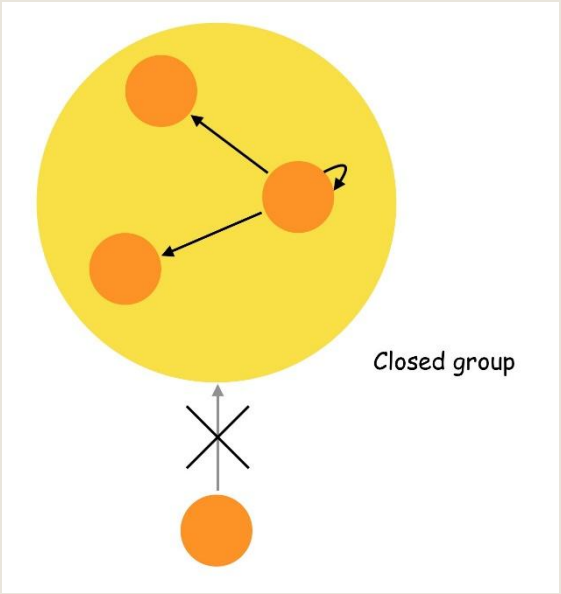
- Group communication is a computing model for multi-party communication that is based on the notion of groups.
  - A group is a set of parties that want to exchange information in a reliable and consistent manner.
- There are three basic types of group communication.
  - One to many (single sender and multiple receivers)
    - It is also called multicast communication.
  - Many to one (multiple senders and single receiver)
  - Many to many (multiple senders and multiple receivers)
- Why Group Communication ?
  - Highly available servers (client-server), Database Replication
  - Multimedia Conferencing, Online Games, ….

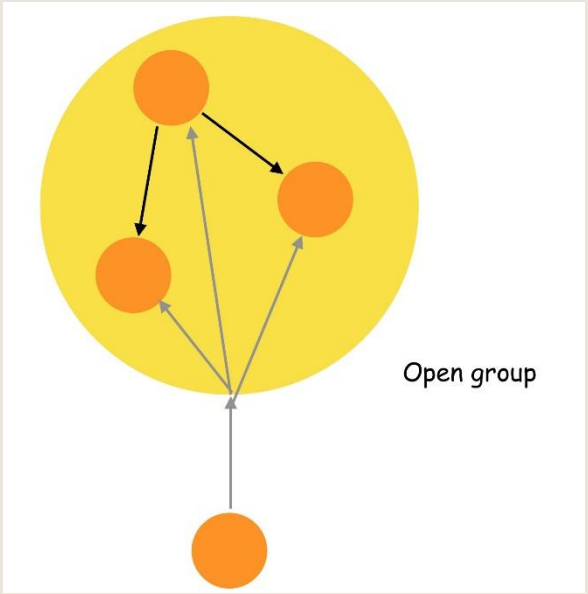
# High Available Web Servers



# Open and Closed Groups

- A closed group is one in which only the members of the group can send a message to the group.
  - An outside process cannot send a message to the group as a whole, although it may send a message to an individual member of the group.
- An open group is one in which any process in the system can send a message to the group.



# Group Membership Management

- A group communication facility should provide flexible primitives
  - to create and delete groups dynamically and
  - to allow a process to join or leave a group at any time.
- A centralized group server (CGS) may be used to manage the groups and their membership information.
  - The CGS maintains a list of member process IDs for each group.
    - All requests to create a group, to delete a group, to add a member to a group, or to remove a member from a group are sent to the CGS.
  - Cons: single point of failure and performance bottleneck.
  - Pros: easy to implement.

# Group Addressing

- A two-level naming scheme is normally used for group addressing.
  - The high-level group name is an ASCII string that is independent of the location information of the processes in the group.
  - The low-level group name depends on the underlying network.
    - Multicast enabled network: a multicast address can be used as a low-level name.
      - A packet sent to a multicast address is automatically delivered to all machines listening to the address.
    - Broadcast enabled network: the broadcast address may be used as a low-level name.
      - A packet sent to the broadcast address is automatically delivered to all host.
      - Each host must check to see if the packet is intended for it.
        - If not, the packet is simply discarded.
      - All groups have the same low-level name, the broadcast address.
    - The low-level name contains a list of hosts that have a process in the group.
      - The sending host sends the message separately to each host in the list.

# Message Delivery in Multicasting

- A sender sends a message to a group by specifying its high level name.
  - The kernel of the sending host contacts the CGS to obtain the low level name and the list of PIDs of the group.
    - This list of PIDs is then embedded into the message as part of the header.
  - The kernel either sends the message to the multicast/broadcast address or sends the message separately to each host in the host list.
- When the message reaches a host, the kernel of that host extracts the list of PIDs and forwards the message to those processes in the host.

# Buffered / Unbuffered Multicasting

- Multicasting is an asynchronous communication mechanism.
  - It is unrealistic to expect a sending process to wait until all the receiving processes that belong to the multicast group are ready to receive the multicast message.
  - The sending process may not be aware of all the receiving processes that belong to the multicast group.
- How a multicast message is treated on a receiving side depends on whether the multicast is buffered or unbuffered.
  - For an unbuffered multicast, the message is received only by those processes of the multicast group that are ready to receive it.
    - The message is dropped if the receiving process is not ready to receive it.
  - For a buffered multicast, the message is buffered for the receiving processes, so all processes will eventually receive the message.

# Semantics of Multicasting

- There are two kinds of semantics in multicast.
- Send-to-all semantics.
  - A copy of the message is sent to each process of the multicast group and the message is buffered until it is accepted by the process.
- Bulletin-board semantics.
  - A message to be multicast is addressed to a channel instead of being sent to every individual process of the multicast group.
    - The channel plays the role of a bulletin board.
  - A receiving process copies the message from the channel instead of removing it when it makes a receive request on the channel.
  - All processes that have receive access right on the channel constitute the multicast group.

# Reliability in Multicasting

- The sender of a multicast message can specify the number of receivers from which a response message is expected.
- In multicast communication, the degree of reliability is normally expressed in the following forms:
  - The 0-reliable. No response is expected from any of the receivers.
    - An example of this type of application is a time signal generator.
  - The l-reliable. Only one response is expected from any of the receivers.
    - Sends a request to several agents and selects the fastest for the future request.
  - The M-out-of-N-reliable.
    - The multicast group consists of N receivers and the sender expects M responses from receivers, where 1 <M <N.
    - Erasure coding, RAID, etc.
  - All-reliable. The sender expects a response from all the receivers.

# Atomic Multicast

- Atomic multicast has an all-or-nothing property.
  - An atomic multicast message is either
    - received by all the processes in the group or
    - it is not received by any of them.
- Only all-reliable applications need atomic multicast facility.
- A simple method of implementing atomic multicast is by using timeouts, ACKs and retransmissions.
  - The method works fine as long as the sending host and all receiving hosts do not fail during an atomic multicast operation.

# Fault Tolerant Atomic Multicast

- A simple FT Atomic Multicast algorithm is as follows.
- The sender sends a message to all members of the group.
  - Timers are set and retransmissions are sent when necessary.
- When a process receives a message,
  - if it has not yet seen this particular message, it sends the message to all members of the group.
    - Timers are set and retransmissions are sent if necessary.
  - If it has already seen the message, the message is discarded.
- It can be shown that no matter how many machines crash or how many packets are lost, eventually all the surviving processes will get the message.

# Multicast Primitives

- Multicast communication facility is usually implemented as a middleware that provides a set of primitives to applications.
- Most systems provide a separate primitive, such as send_group (), for sending a message to a group.
  - Send () is then used only for one to one communication.
- This simplifies the design and implementation of a group communication facility.
  - Name server is used by send () for two-level process naming.
  - Group server is used by send_group () for two-level group naming.
    - An extra parameter may be used to specify the degree of reliability.
    - A 2nd extra parameter can be used to specify the atomicity requirement.

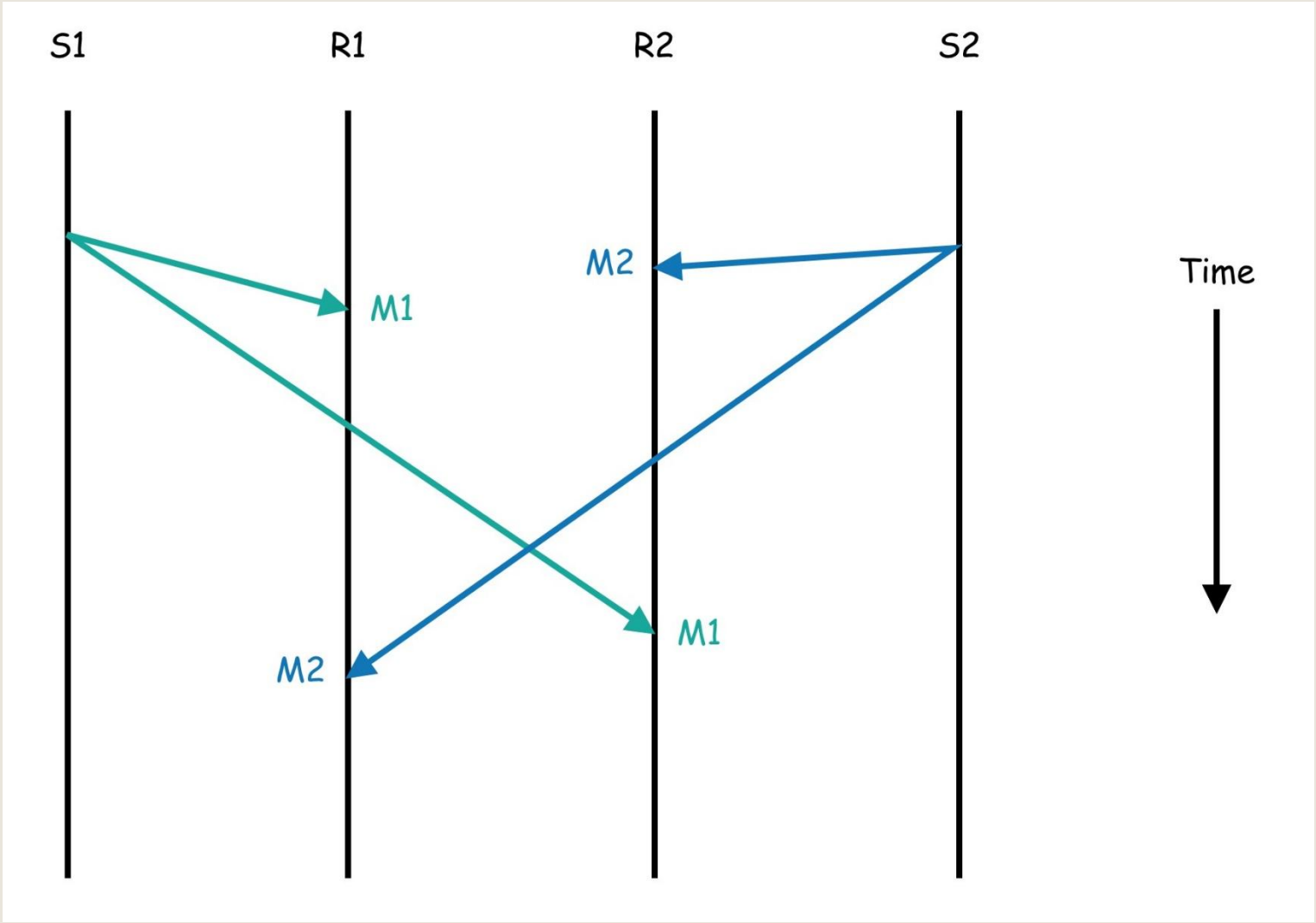
# Many to One Communication

- Multiple senders send messages to a single receiver.
- The single receiver may be selective or nonselective.
  - A selective receiver specifies a unique sender.
    - A message exchange takes place only if that particular sender sends a message.
  - A nonselective receiver specifies a set of senders.
    - If any sender in the set sends a message to this receiver, a message exchange then takes place.
- An important issue related to the many-to-one communication scheme is nondeterminism.
  - A receiver may want to wait for information from a sender that it is not known in advance.

# Many to Many Communication

- Multiple senders send messages to multiple receivers.
  - The multicast and many-to-one are special cases of this scheme.
- One of an important issue for the many to many communication is ordered message delivery.
  - Ordered message delivery ensures that all messages are delivered to all receivers in an order acceptable to the application.
    - Multiple clients updating the same DB that has multiple replicates requires that all update messages be delivered in the same order to all replicates.
  - Ordered message delivery requires message sequencing.
    - In multicasting, a sender can initiate the next multicast only after the previous multicast message has been received by all the members.
    - In many to one, since there is only one receiver, the ordering can simply be handled by the receiver.

# No Ordered Message Delivery

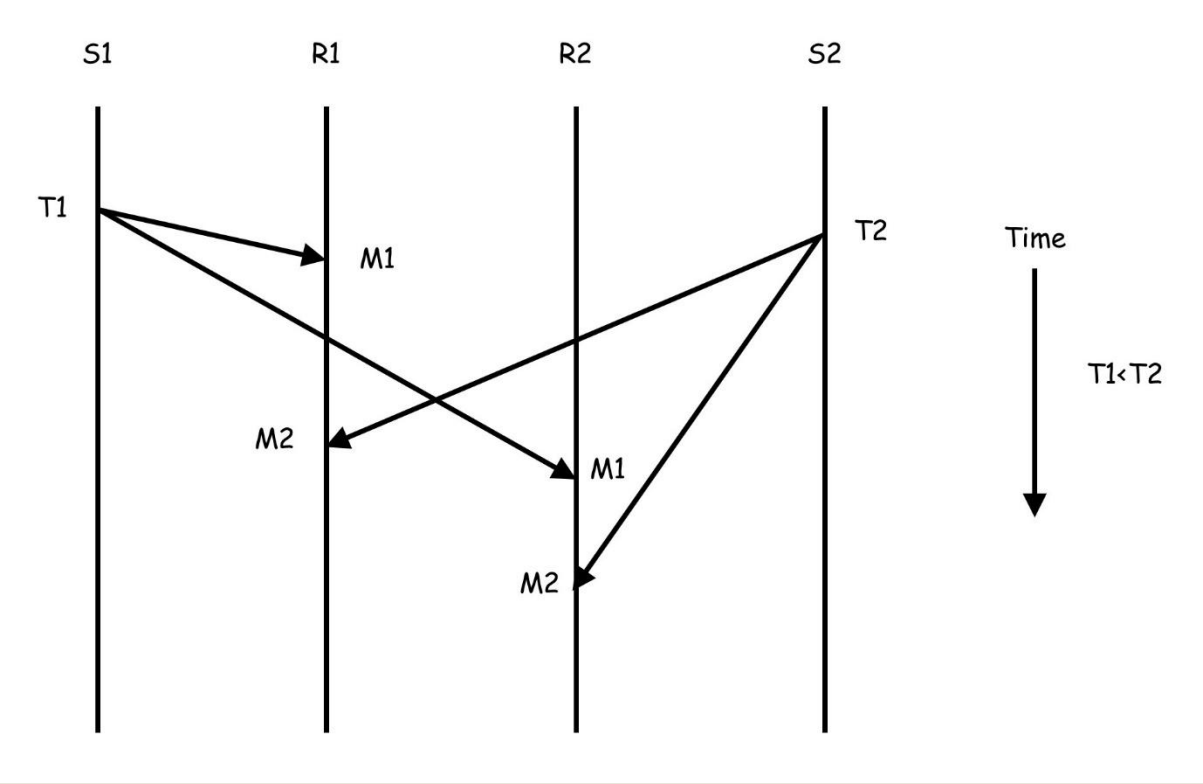


# Semantics of Message Ordering

- Absolute Ordering: all messages are delivered to all receivers in the exact order in which they were sent.
- Consistent Ordering, all messages are guaranteed to arrive at all receivers in the same order, but that order may not be the real order in which they were sent.
- Causal Ordering, it ensures that if the event of sending one message is causally related on the event of sending another message, the two messages are delivered to all receivers in the same causal order.
  - If, however, two messages are concurrent (not causally related), no guarantees are made, and the system is free to deliver them in a different order to different processes if this is easier.

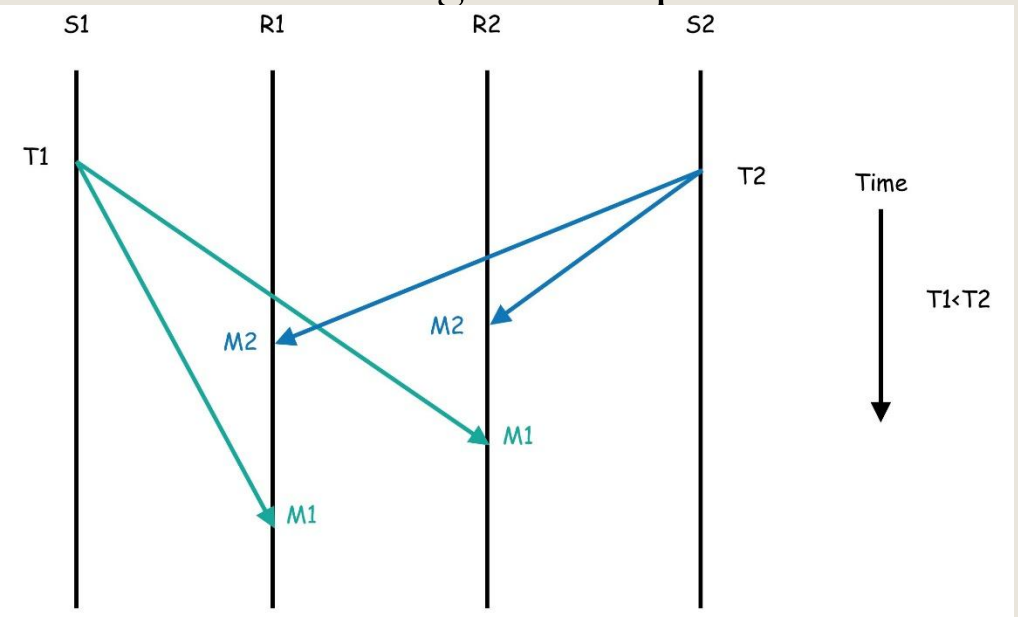
# Absolute Ordering

- One method to implement absolute ordering is to attach each message a globally synchronized timestamp (sync clocks).
  - All receiving host delivers messages to its processes in the order of timestamps.



# Consistent Ordering

- Consistent ordering may be implemented by combining many-to-one and one-to-many schemes.
  - A sending host 1st sends a control message to a special receiver (known as a sequencer) that responses a unique seq#.
  - The sending host then multicasts the real message with seq#.
  - All receiving host delivers messages to its processes in the order of seq#.



# The ABCAST Protocol (1)

- The ABCAST protocol of the ISIS system is a distributed sequencing algorithm that provides consistent ordering.
  - The sender assigns a temporary sequence# to the message and sends it to all the members of the group.
    - The temp seq# must be greater than any previous used seq# by the sender.
  - On receiving the message, each receiver returns a proposed seq# to the sender.
    - The member (i) calculates its proposed seq# by using the function

      $$\mathbf{max(F_{max}, P_{max}) + 1 + (i/N)}$$

      - $F_{max}$ is the largest final seq# agreed upon so far by the group.
      - $P_{max}$ is the largest proposed seq# by the member (i).

# The ABCAST Protocol (2)

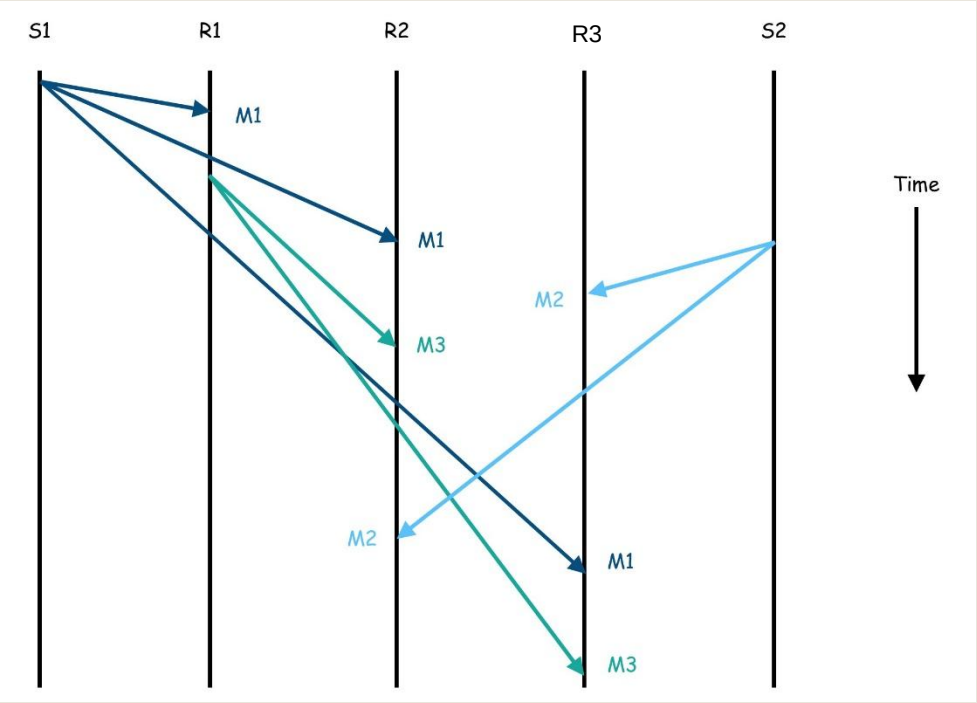
- On receiving all proposed seq#, the sender selects the largest one as the final seq# for the message and sends it to all members in a commit message.
  - The chosen final seq# is guaranteed to be unique because of the term (i/N) in the function of calculating proposed seq#.
- On receiving the commit message, each receiver attaches the final seq# to the message.
- Committed messages with final seq# are delivered to the applications in the order of their final seq#.

# References


- Birman, K. P., "The Process Group Approach to Reliable Distributed Computing," Communications of the ACM, Vol. 36, pp. 36-53 (1993).
- Birman, K. P., Schiper, A., and Stephenson, P., "Lightweight Causal and Atomic Group Multicast," ACM Transactions on Computer Systems, Vol. 9, No.3, pp. 272-314 (1991).
- Birman, K. P., and Joseph, T. A., "Reliable Communication in the Presence of Failures," ACM Transactions on Computer Systems, Vol. 5, No.1, pp. 47-76 (1987).

# Causal Ordering

- One weaker ordering semantics that is acceptable to many applications is the causal-ordering semantics.
  - m1→m3, m3 depends on m1
  - m1||m2, m1 and m2 are concurrent
  - m2||m3, m2 and m3 are concurrent

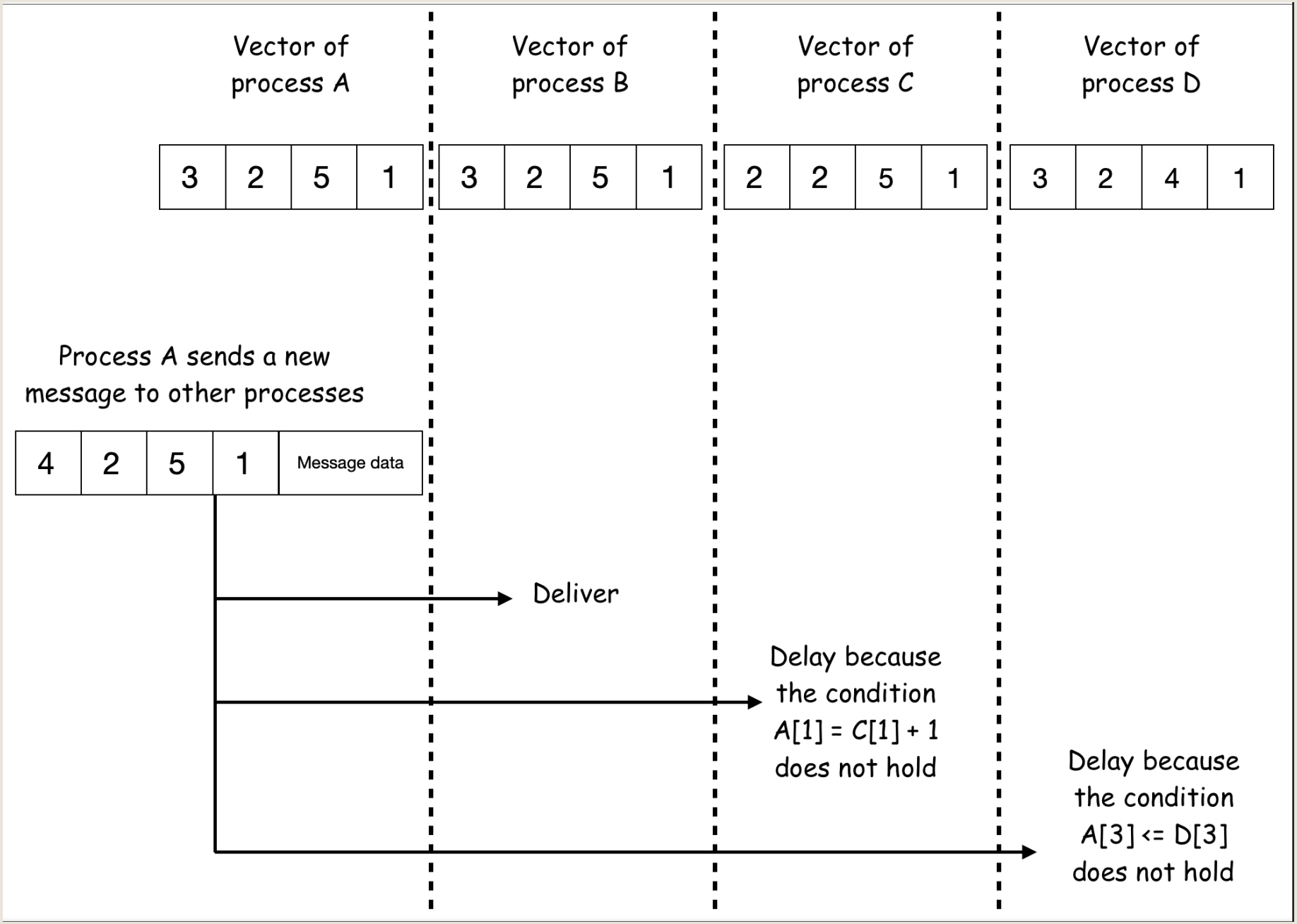


# The CBCAST Protocol (1)

- Each member of a group maintains a vector of N elements.
  - The $i^{th}$ element records the largest seq# received by this member from the member i.
- To send a message, a process $1^{st}$ increments the seq# of its own element in its own vector and then sends the message with the vector.
- On receiving a message, the receiving host $1^{st}$ puts it in a buffer.
  - Let S be the vectors attached on the received message.
  - Let R be the vector maintained by the receiver itself.
  - Let i be the unique ID of the sender.
  - If $S[i]=R[i]+1$ and for all $j \neq i$, $S[j] \leq R[j]$, then the message is delivered to the application, otherwise, the message is left in the buffer.
  - $R[i] \leftarrow S[i]$

# The CBCAST Protocol (2)

- Periodically, each undelivered message is tested to decide whether to deliver it to the application or remain in the buffer.
  - Let the undelivered message is sent from the member i.
  - If $S[i]=R[i]$ and for all $j\neq i$, $S[j]\leq R[j]$, then the message is delivered.
- The test of $S[i]=R[i]+1$ is to ensure that the receiver has not missed any message sent from the sender i.
  - Two messages from the same sender are always causally related.
- The test of for all $j\neq i$, $S[j]\leq R[j]$ is to ensure that the sender has not received any message that the receiver has not yet received.
  - This test is needed to make sure that the sender's message is not causally related to any message missed by the receiver.


Vector of process A
Vector of process B
Vector of process C
Vector of process D
3 2 5 1
3 2 5 1
2 2 5 1
3 2 4 1
Process A sends a new message to other processes
4 2 5 1 Message data
Deliver
Delay because the condition A[1] = C[1] + 1 does not hold
Delay because the condition A[3] <= D[3] does not hold