Message Passing

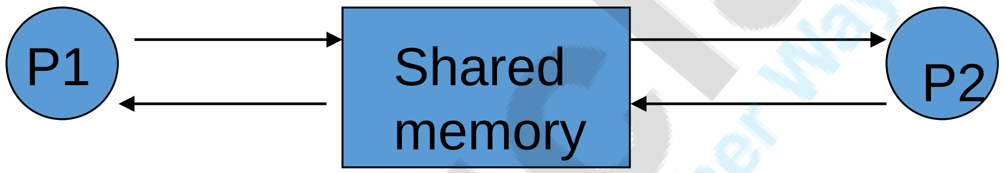
INTER – PROCESS COMMUNICATION

Inter Process Communication

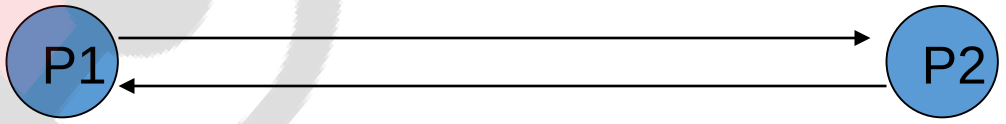
- Inter process Communication is the heart of any distributed system
- IPC means two processes running on two different computers are in communication with each other
- Hence IPC requires information sharing among two or more processes
- The two basic methods of sharing are
  - Original sharing, or shared data approach
    - In this approach, the information to be shared is placed in a common memory area that is accessible to all the processes involved in IPC. This is depicted pictorially in the next slide
  - Copy sharing, or message passing approach
    - In this approach, the information to be shared is physically copied from sender process's address space to the address spaces of all the receiver processes and this is done by transmitting the data to be copied in the form of messages (message is a block of information). The picture shows the message passing paradigm. That is the communicating processes interact directly with each other

Inter Process Communication(Cont'd)

- Original sharing (shared-data approach)



- Copy sharing (message passing approach)
  - Basic IPC mechanism in distributed systems
  - It enables by using simple communication primitives like *send* and *receive* for the programs to exchange messages



Desirable Features Of A Good Message Passing System

Desirable Features of a Good MPS

- Simplicity
  - It should be simple and easy to use
  - It should be straightforward to construct new applications and communicate with existing ones by using primitives provided by the MPS
- Uniform Semantics
  - In a distributed system, a MPS may be used for the following two types of IPC
    - Local communication, in which communicating processes are on the same node

Desirable Features of a Good MPS (Cont'd)

- Remote communication, in which communicating processes are on different nodes
- One important issue in the design of MPS is that the semantics for local and remote communication should be as close to each other as possible
- This make the system easy and seamless to implement

- Efficiency
  - MPS can be made efficient by reducing the no. of message exchanges
  - Some of the optimization techniques used are:
    - Avoid the costs of establishing and terminating connections between the same pair of processes
    - Minimize the cost of maintaining the connections
    - Piggybacking of acknowledgement of previous message with the next message

Desirable Features of a Good MPS (Cont'd)

- Reliability
  - DCS are prone to catastrophic events like node crashes and communication link failure resulting in loss of messages
  - A reliable IPC will Cope with failure problems & guarantee delivery of messages
  - This is normally done by acknowledgements and retransmissions on the basis of timeouts
  - Another issue is duplicate messages because of failures and timeouts
  - A good IPC is capable of detecting & handling of duplicate messages by using sequence numbers to messages

Desirable Features of a Good MPS (Cont'd)

- Correctness
  - A good IPC has protocols to handle group communications that allow a sender to send messages to a group receivers and a receiver to receive messages from several senders. Issues related to correctness are as follows
    - Atomicity: ensures that group message is delivered to all of them or none of them
    - Ordered Delivery: ensures that the messages arrive at all receivers in the order acceptable to the application
    - Survivability: guarantees the delivery of the messages even with partial failure of processes, systems or communication links. This is more difficult to achieve

Desirable Features of a Good MPS (Cont'd)

- Flexibility
  - Not all users require the same level of reliability and correctness of IPC
  - Many application do not require atomicity or ordered delivery of messages
  - Hence the IPC should have flexibility to choose & specify type & level of reliability & correctness requirement
  - It should be flexible enough to permit any kind of control flow between the cooperating processes, including synchronous and asynchronous *send/receive*
- Security
  - A good MPS must also be capable of providing a secure end to end communication

Desirable Features of a Good MPS (Cont'd)

- i.e. message in transit on the network should not be accessible to any user other than those to whom it is addressed and the sender
- Steps necessary for secure communication include:
  - Authentication of the receiver(s) of a message by the sender
  - Authentication of the sender of the message by receiver(s)
  - Encryption of a message before sending it over the network

- Portability
  - Message passing system & applications using it should be portable
  - There are two different aspects of portability in MPS

Desirable Features of a Good MPS (Cont'd)

- The message passing system should itself be portable
  - i.e., it should be possible to easily construct a new IPC facility on another system by reusing the basic design of the existing MPS
- The Applications written by using the primitives of the IPC protocols of an MPS should be portable
  - This requires the hardware/software heterogeneity to be addressed while designing MPS
  -

Issues in IPC
By
Message Passing

What is a Message?

- A message is a block of information formatted by a sending process in such a manner that it is meaningful to the receiving process
- It consists of fixed size header and variable size collection of data objects
- As shown in the next fig. the header consists of following elements
  - Address: It contains characters that uniquely identify the sending and receiving processes in the network. It has two parts, the sender address and receiver address
  - Sequence number: The message identifier, which is very useful in identifying lost or duplicate messages in case of system failures
  - Structural Information: This has two parts;
    - The type part specifies whether the data is embodied in the message or it contains only a pointer to the data, which is stored somewhere outside the contiguous portion of the message.
    - The second part specifies the length of the variable size message data

Message Structure

- In a message oriented IPC protocol, the sending processes determines contents of a message and the receiving process is aware of how to interpret the contents
- Special primitives are explicitly used for sending and receiving the messages
- Hence the users are fully aware of the message formats used in the communication process and the mechanisms used to send and receive messages

Actual data or pointer to the data	Structural information		Sequence number or message ID	Addresses	
	Number of bytes /elements	Type(Actual data or pointer to data)		Receiving process address	Sending process address

$\underbrace{\text{Actual data or pointer to the data}}_{\textbf{Variable size data}}\ \underbrace{\text{Structural information, Sequence number or message ID, Addresses}}_{\textbf{Fixed length header}}$

Issues in IPC by Message Passing

- In the design of an IPC protocol for an MPS, the following important issues need to be addressed
  - Who is the sender/receiver?
  - Is there one receiver or many receivers?
  - Is the message guaranteed to be accepted by its receiver(s)?
  - Does the sender need to wait for a reply?
  - What should be done if a catastrophic event such as a node crash or a communication link failure occurs during the course of communication?
  - What should be done if the receiver is not ready to accept the message: will the message be discarded or stored in a buffer? If so what should be done if the buffer is full
  - If there are several outstanding messages for a receiver, can it choose the order in which to service the outstanding messages?
- These issues are addressed by the semantics of the set of communication primitives provided by the IPC protocol

Synchronization

Synchronization

- A central issue in the communication structure is the synchronization imposed on the communicating processes by the communication primitives
- Synchronization imposed on the communicating processes basically depends on two types of semantics used by the send and receive primitives
  - Nonblocking: If its invocation does not block the execution of its invoker (the control almost immediately returns to the invoker)
  - Blocking: Else it is called blocking type
- The two types of semantics that are used on both send & receive primitives
  - In case of blocking send primitive, after execution of the send statement, the sending process is blocked until it receives an acknowledgement from the receiver

Synchronization

- In case of nonblocking send primitive, after the execution of the send statement, the sending process is allowed to proceed with the execution, as soon as the message is copied to a buffer
- In case of blocking receive primitive, after execution of the receive statement the receiving process is blocked until it receives a message
- In case of nonblocking receive primitive, the receiving process proceeds with its execution after execution of the receive statement

- Complexities in synchronization
  - An important issue in a nonblocking receive primitive, is how the receiving process knows when message has arrived in message buffer in non blocking receive? One of the following method is commonly used
    - Polling: In this method a test primitive is provided to allow the receiver to check the buffer status. The receiver uses this primitive to periodically poll the kernel to check status of the message buffer

Synchronization

- Interrupt: When the buffer is filled with a message and ready for use by the receiver, a software interrupt is used to notify the receiving process
- This avoids frequent polling with test primitive
- A variant of nonblocking receive primitive is the conditional receive primitive, which also returns the control to the receiving process immediately, either with a message or with an indicator that no message is available

- **In a Blocking send/receive, sender/receiver could get blocked forever if receiver/sender crashes or message is lost.**

Synchronous vs. Asynchronous Communication

- Timeout: To avoid this send primitive commonly uses a time out value that specifies a time interval after which the send operation is terminated with an error status
- A timeout value may also be associated with a blocking receive primitive to prevent the receiving process from getting blocked indefinitely because of sending process has crashed or communication failure

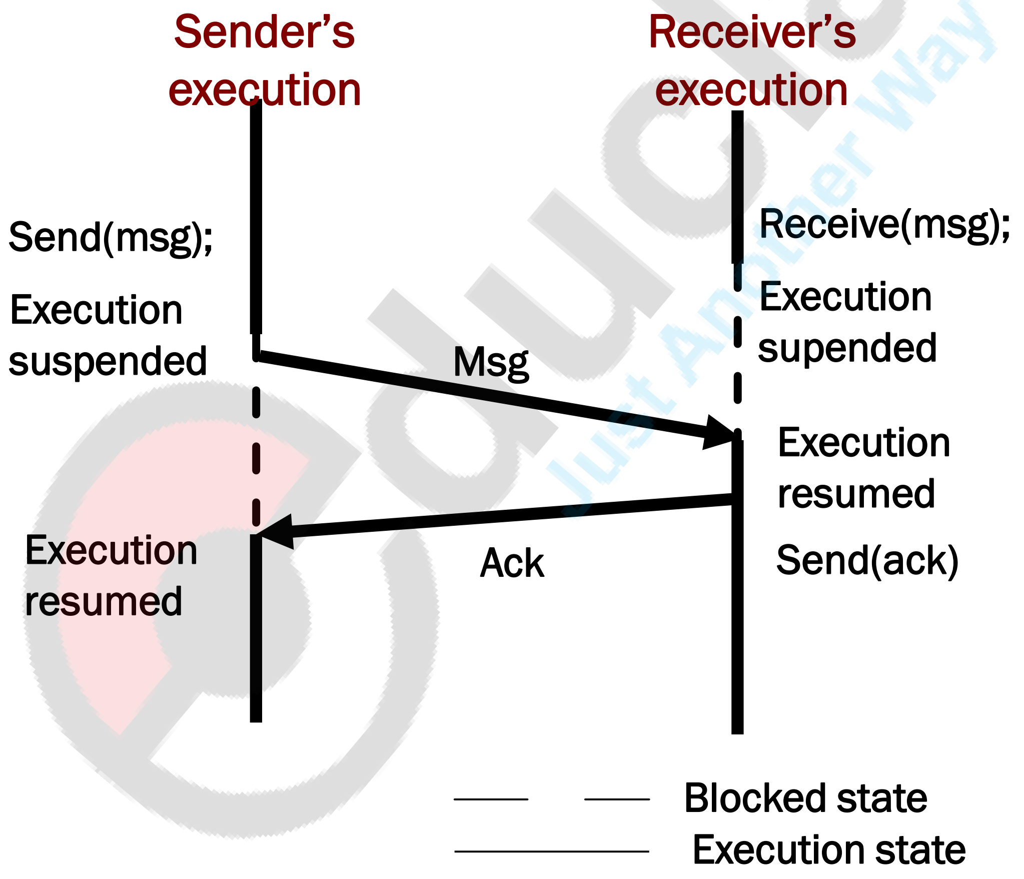
- When both send and receive primitives of a communication between two processes use blocking semantics, the communication is said to be Synchronous Communication else it is asynchronous
  - For a synchronous communication, the sender and receiver must be synchronized to exchange a message.

Synchronous Communication

- Advantages
  - Simple & easy to implement
  - Reliable
- Disadvantages
  - Limits concurrency
  - Can lead to communication deadlock
  - Less flexible as compared to asynchronous
  - Hardware is more expensive

Synchronous Communication

- When both send and receive primitives use blocking semantics



Asynchronous Communication

- Advantages
  - Doesn't require synchronization of both communication sides
  - Cheap, timing is not as critical as for synchronous transmission, therefore hardware can be made cheaper
  - Set-up is very fast, well suited for applications where messages are generated at irregular intervals
  - Allows more parallelism
- Disadvantages
  - Large relative overhead, a high proportion of the transmitted bits are specially for control purposes and thus carry no useful information
  - Not very reliable
- A flexible message passing system usually provides both blocking and unblocking primitives for send and receive, so that users can choose the most suitable one to match the specific needs of their applications

Buffering

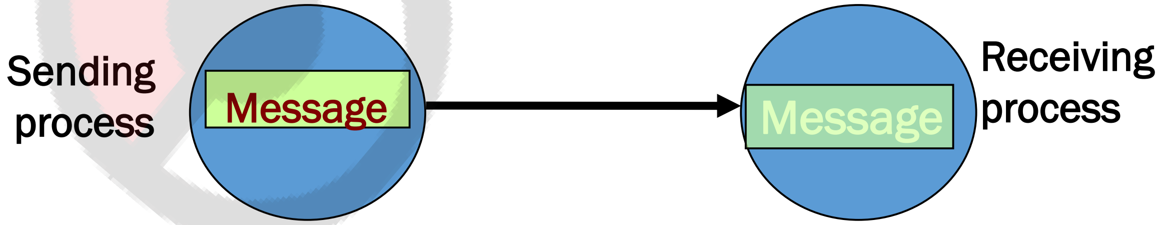
- Message transmission from one process to another by copying the body of the message from the address space of sending process to the address space of the receiving process
    - This may use address spaces of the kernel of the sending and receiving computers
- In case the receiving process is not ready to receive the message, but would like the OS to save the message for later reception
- OS utilizes the message buffer space for this purpose for interim storage of the messages until the receiving process executes the specific code to receive the message
- In IPC, the message buffering strategy is strongly related to synchronization strategy

Buffering (Cont'd)

- Synchronous and asynchronous modes of communication correspond respectively to the two extremes of buffering: namely a null buffer or no buffer and a buffer with unbound capacity
- Other two commonly used buffering strategies are single message and finite bound or multiple message buffers
- Each of them have their own advantages and disadvantages
- The selection of the buffering mode is driven by the application requirements

Null Buffer (No Buffering)

- In case of no buffering, there is no place to temporarily store the message
- Therefore one of the two following implementation strategies are used
- The message remains in the sender process address space, and send is delayed until the receiver executes a corresponding receive code
  - To do this the sender process is blocked up and suspended in such a way that when it is unblocked, it starts by re-executing the send statement
  - When the receiver executes receive, an acknowledgement is sent to the sender's kernel saying that the sender can now send the message

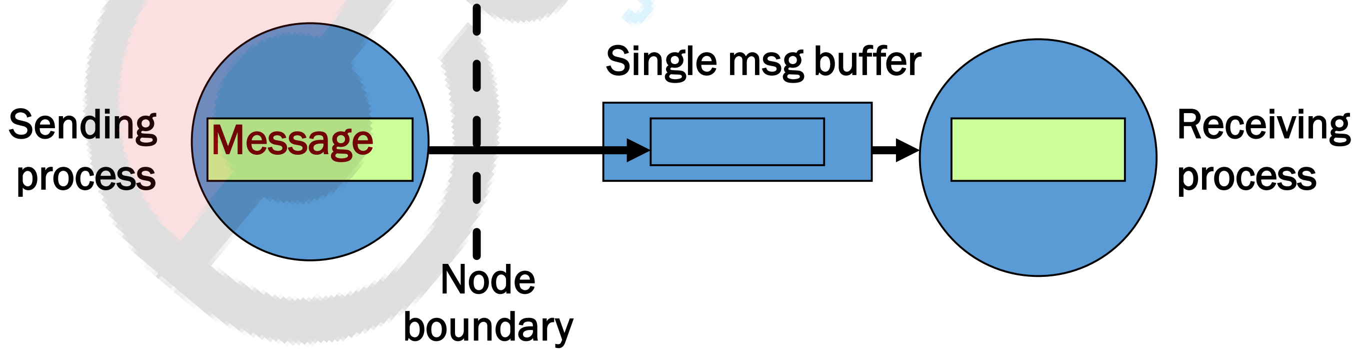


Null Buffer (Cont'd)

- In other method the timeout mechanism is used to resend the message after a time out period
  - i.e. after executing the *send* the sender waits for an ACK from the receiver process
  - As shown in the figure the message transfer is directly from the sender's address space to the receiver's address space
  - The *send* gives up after pre-decided number of tries
- The null strategy is generally not suitable for synchronous communication between two processes in DCS because if the receiver is not ready, the message is resent several times
- The receiver has to wait for entire time taken to transfer the message across the network which may be significant

Single Message Buffer

- Hence the synchronous communication mechanisms in network / distributed systems use a single message buffer strategy
- The idea behind the single buffer strategy is to keep the message ready for use at the location of the receiver
- This is because in systems based on synchronous communication, an application module may have at most one message pending at a time
- The buffer may be in the receiver kernel or process address space
- The fig shows the two stage message passing process



Unbounded Capacity Buffer

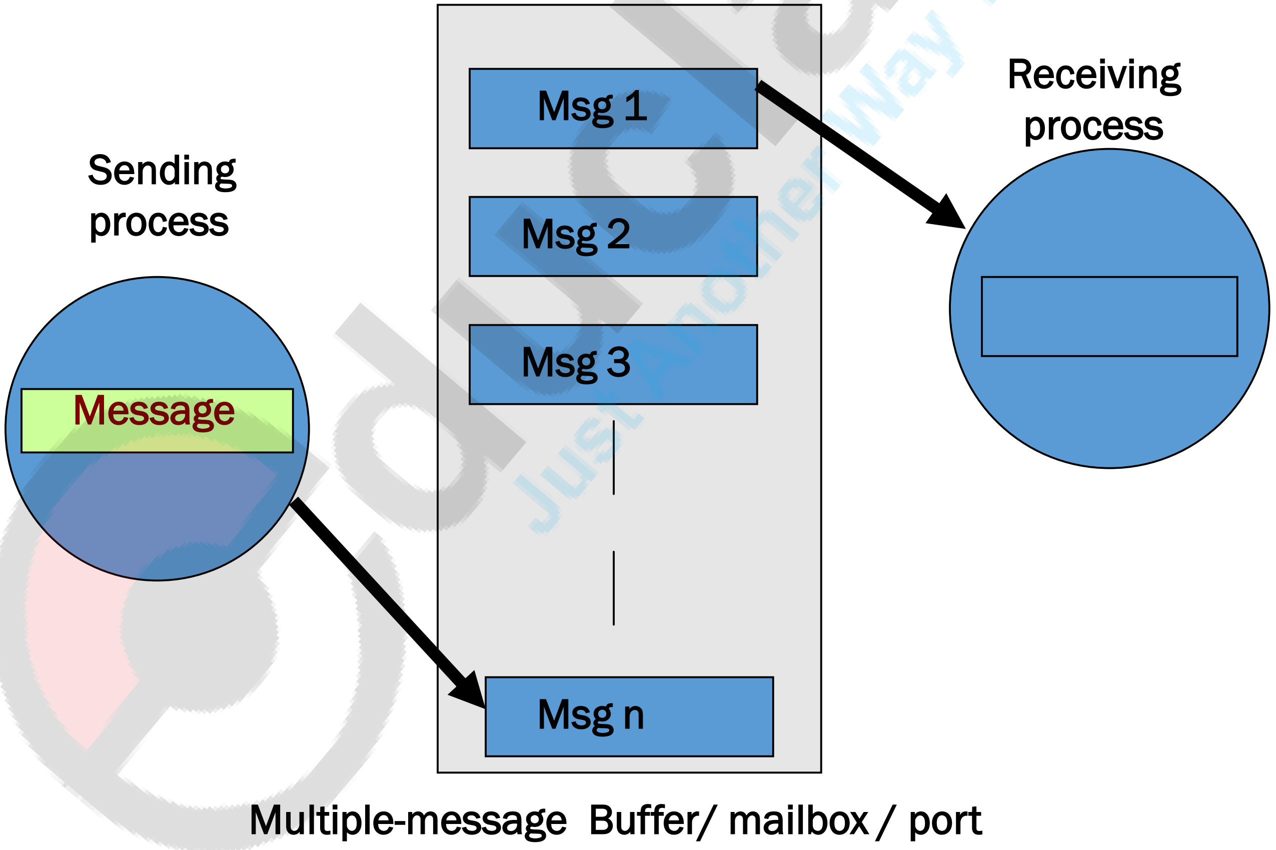
- In the asynchronous mode of communication, since the sender does not wait for the receiver to be ready, there may be several pending messages that have not yet been accepted by receiver
- Hence an unbounded capacity of message buffer that can store all not received messages is needed to support asynchronous communication with the assurance that all the message sent to receiver will be delivered
- Unbounded capacity of a buffer is practically impossible as the memory available is finite

Finite Bound (Multiple Message) Buffer

- In practice asynchronous communication use finite bound buffers also known as multiple message buffers
- Hence a strategy has to be in place to address buffer overflow
  - Unsuccessful Communication: In this method, message transfers simply fail when buffer is full
    - The *send* normally sends an error message back to the sender as buffer is full which makes message passing less reliable
  - Flow controlled communication: the second method is to use flow control which means that the sender is blocked until the receiver accepts some messages, thus clearing buffer space
    -

Finite Bound Buffer (Cont'd)

- This method is used in asynchronous communication.



Multidatagram Messages

- All Networks have an upper bound on the size of the data that can be transmitted across a network
- This size is known as Maximum transfer unit (MTU) of a network
- A message whose size is larger than MTU has to be fragmented into multiples of MTU and then each fragment has to be sent separately
- Each Packet consisting of Message data + control information is called a datagram
- Messages with smaller size are sent as single packet and known as single-datagram messages

Multidatagram Messages (Cont'd)

- Messages with multiple fragments are known as multidatagram messages
- Different packages of a multidatagram message bear a sequential relationship to one another
- Disassembling on the sender side and reassembling in sequence, of packets of multidatagram messages, on the receiver side is usually the responsibility of the message passing system
- This is major responsibility as the messages may not be received in the order it is sent and re-sequencing of messages identifying if any in between message is missing etc., is an important function
- How this is handled depends on MPS implementation

Encoding and Decoding of message data

- Due to the problems discussed below in transferring program objects in their original form, they are converted to a stream form that is suitable for transmission and placed into a message buffer
  - **Different program objects occupy varying amount of address space.**
  - **An absolute pointer value loses its meaning when transmitted from one process address space to another**
- This conversion of the data takes place on the sender and is known as encoding of a message data

Encoding and Decoding of message data (Cont'd)

- The encoded message when received by the receiver, it must be converted from the stream form back to the program object before it can be used
- This process of reconstruction of the program objects from the message data is known as decoding of message data
- One of the following two representations may be used for the encoding and decoding of message data
  - In *tagged representation*, the type of each program object along with its value is encoded in the message
  - In this method, it is simple matter to the receiving process to check the type of each program object in the message because of the self-describing nature of the coded data format

Encoding and Decoding of message data (Cont'd)

- Quantity of data transferred is more. Time taken to encode/ decode data is more
- In *untagged representation*, the message data only contains the program objects
- No information is included in the message data to specify the type of each program object
- In this method, the receiving process should have a prior knowledge of how to decode the received data because the code data is not self describing
- The untagged representation is used in SUN XDR (eXternal Data Representaion) and Courier
- Where as tagged representation is used in the ASN and Mach distributed operating system

Encoding and Decoding of message data (Cont'd)

- In general tagged representation is more expensive than untagged representation both in terms of amount of data transferred as well as processing time required for encoding and decoding of message data
- No matter what method is used, the sender as well as the receiver should be fully aware of the format of data coded in the message
- The sender process uses an encoding routine to convert data into coded form and the receiver uses a decoding routine to generate the original data from the received data

Encoding and Decoding of message data (Cont'd)

- The encoding and decoding processes are symmetrical operations in nature because decoder reproduces the exact data that was encoded allowing for the differences in the data representation in the two systems (Heterogeneous nature of the systems)
- In case receiver receives badly encoded data, it sends an error back saying the encoding not intelligible forcing the sender to regenerate the encoded data and resend it to the receiver
- Hence encoding and decoding of data form an important part of Message Passing System in a Distributed System

Process Addressing

- Another important issue in message based communication is addressing (naming) of the parties involved in an interaction
- To whom does the sender would like to send a message and from whom the receiver wish to accept a message
- The MPS usually supports two types of process addressing
  - Explicit addressing: The process with which communication is desired is explicitly named as a parameter in the communication primitive used
    - Send (process_id , msg): send a message to a process identified by process_id
    - Receive (process_id , msg): receive a message from a process identified by process_id

Methods for Process Addressing

- Implicit addressing: The process willing to communicate does not explicitly name a process for communication
  - Send_any (service_id , msg): The sender names a service instead of a process. This type of addressing is used in a client-server communication when the client is not concerned with which particular server out of a set of servers providing the particular service responds. This type of addressing is called functional addressing as the address used in the primitive identifies a service rather than a process
  - Receive_any (process_id , msg): the receiver is willing to accept a message from any sender. It returns process id of the process from which the message was received. This is again useful in client server communications, when the server is meant to service requests of all clients that are authorized to use its service

Methods for Process Addressing (Cont'd)

- Let us now consider the commonly used methods for process addressing
  - A simple method to identify a process is a combination of machine_id and local_id such as machine_id@local_id
- The local_id part is a process identifier or a port identifier of a receiving process, some thing else that uniquely identify a process on a machine
- A process willing to communicate with another machine sends the message in the form machine_id@local_id,
  - The machine_id identifies the receiving machine
  - Local_id is then used by the kernel of the receiving machine to forward the message to the right process for which it is intended
  - Local_id need to be unique to the receiving machine only and can be generated without consulting other machine
  - However, one of the drawback of this method is that it does not allow process migration, i.e., from one machine to another

Methods for Process Addressing

- This can be overcome by using machine_id, local_id, and machine_id
  - The first field identifies the node on which the process is created
  - The second is the local identifier generated by the node on which the process is created
  - The third field identifies the last known location (node) of the process
- During the life time of process the first two fields do not change, the last one may change. This method is known as link-based process addressing
- When the process migrate, a link information (machine_id of the new node) is left on the previous node and on the new node
- A mapping table is maintained by the kernel of the new node for all processes created on another node but running on this node currently

Location Transparent Process Addressing

- Current location of receiving process is sent to sender, which it caches for future use
- Drawbacks
  - Overhead of locating process large if process migrated many times
  - Not possible to locate process if intermediate node in the process migration chain is down
- Both above methods are location non-transparent as one need to specify the machine identifier
- As transparency is one of the main goals of DCS, a location-transparent process addressing mechanism is desirable
- A simple method is to ensure that every process has system wide unique identifier.

Location Transparent Process Addressing

- Two-level naming scheme
  - Each process has a high level machine independent name and low level machine dependent name
  - Name server is used to maintain a mapping table that maps high level names of processes to low level names, and address of the name server is well known across the network
  - A process that wants to send a message, it uses high level name of the receiving process in the communication primitive.
  - Kernel of sending machine obtains low level name of receiving process from name server and also caches it for future use

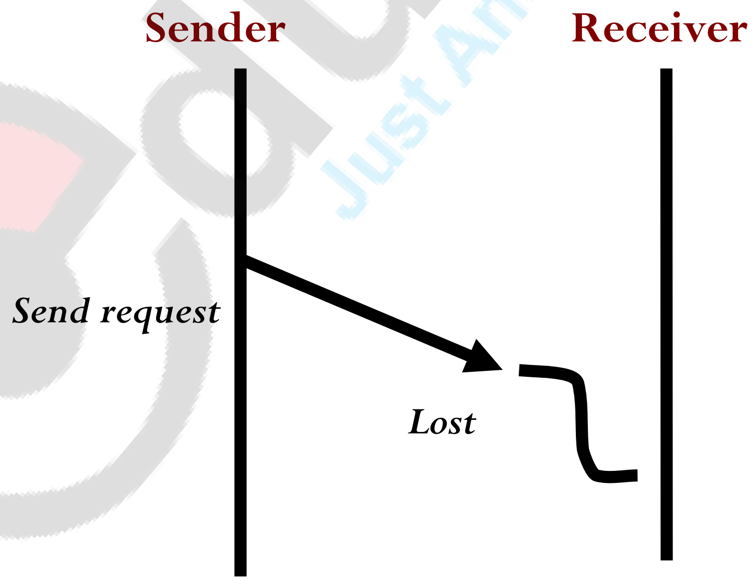
Location Transparent Process Addressing

- The kernel then sends the message using low level name of the receiving process
- The name server approach allows a process to be migrated from one node to another without the need to change the code in the program of any process that wants to communicate with it
- When process migrates only low level name changes and the change is incorporated in name server's mapping table
- This method also suffers from problem of poor reliability and scalability
- One way to overcome this is to maintain replicas of name server to make it scalable and reliable
- However it leads to extra overhead of ensuring that the replicas are always consistent

Failure Handling

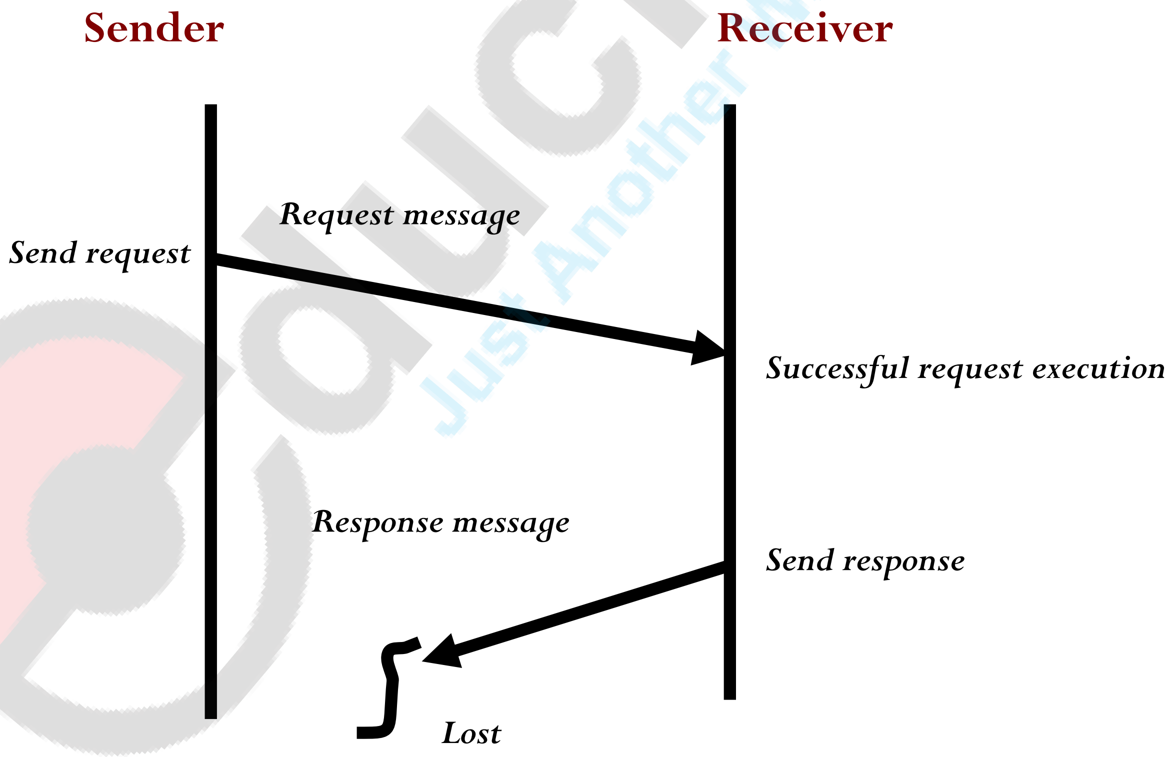
- While DCS provides for parallelism, it is always prone to partial failures like node crash or a communication failure leading to following problems

1. Loss of request msg: This may be due to failure of communication link or receiver node down



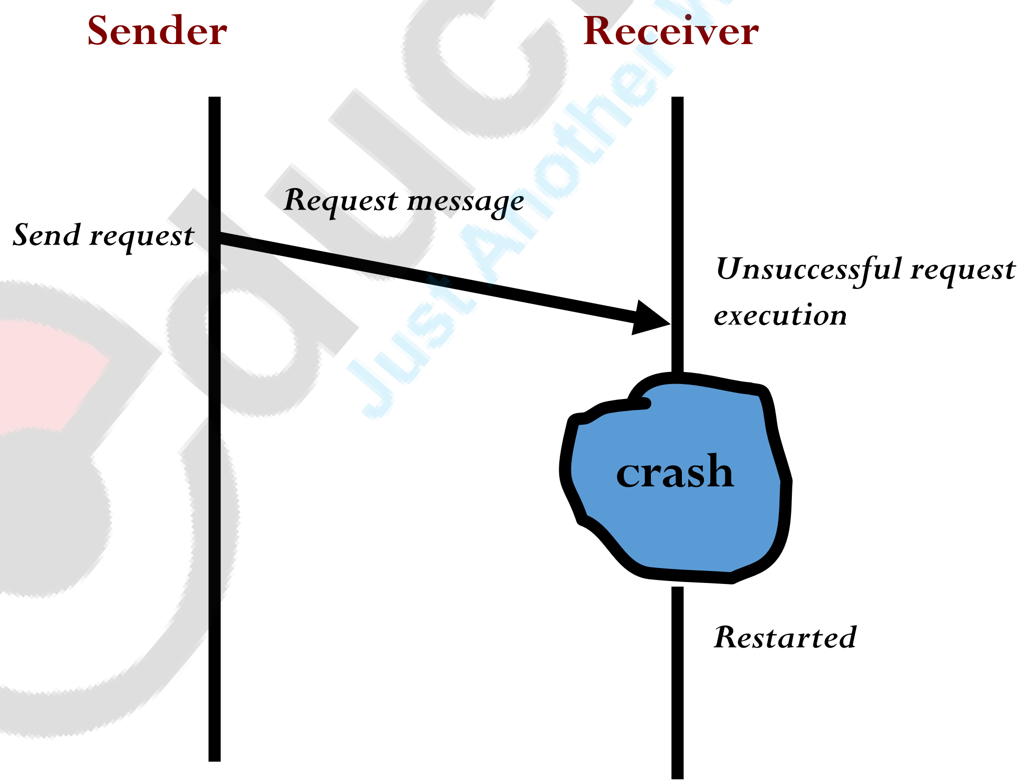
Failure Handling (Cont'd)

2. Loss of response msg: Due to failure of the communication link or the sender machine down when the reply reaches it



Failure Handling (Cont'd)

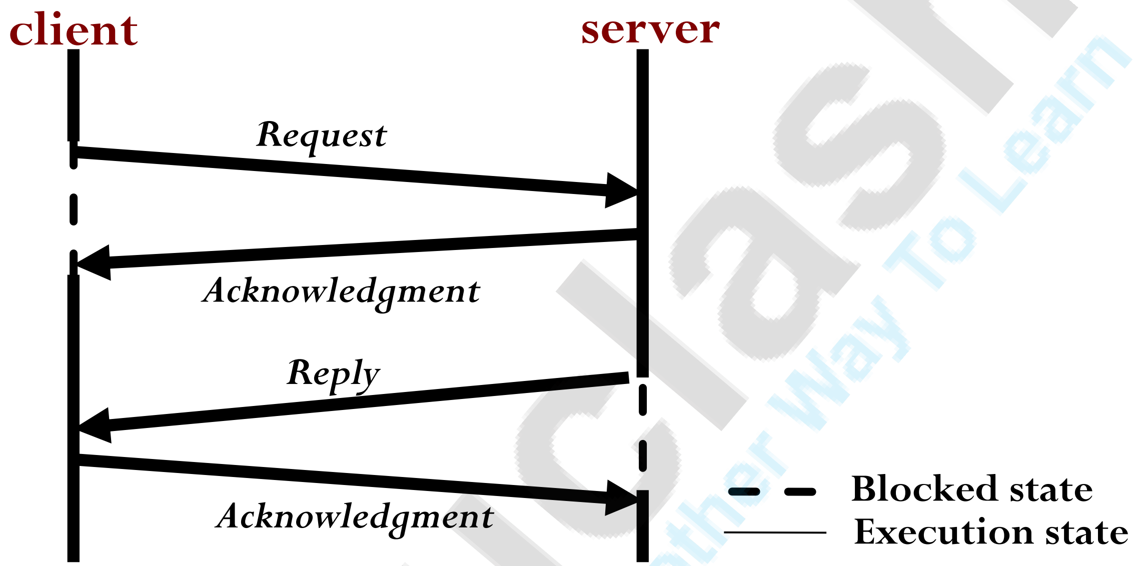
3. Unsuccessful execution of the request: Happens when the receiver crashes while processing the request



Four message reliable IPC protocol

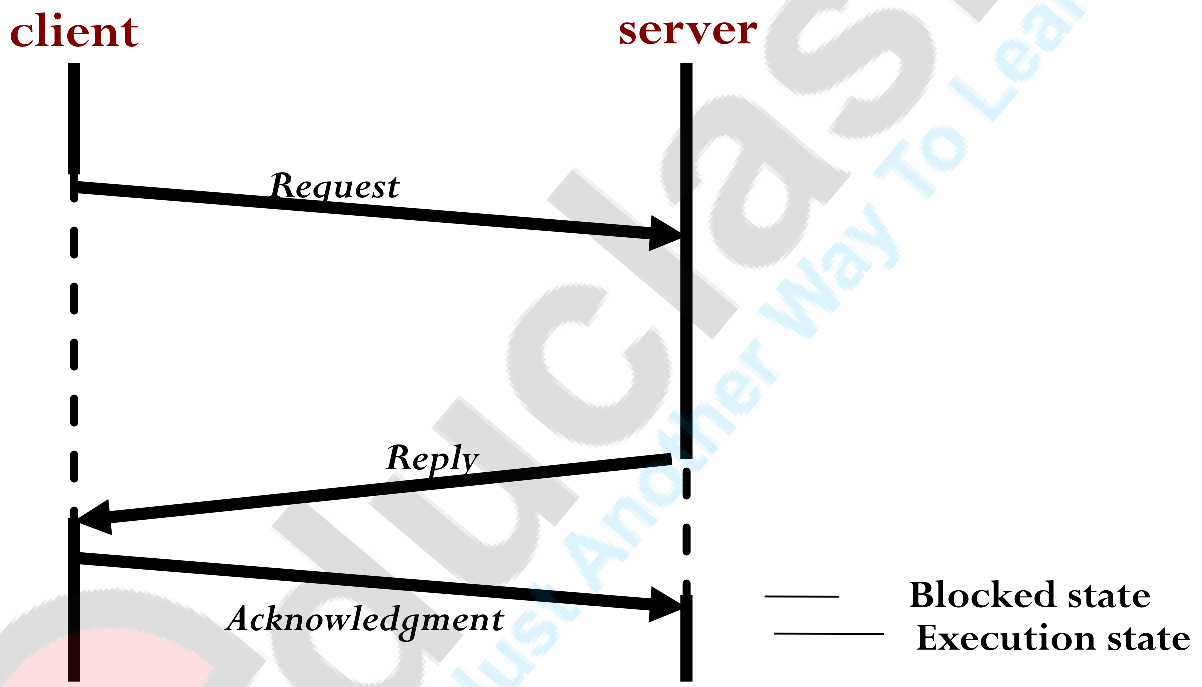
- To cope with this problem, a reliable IPC protocol of a MPS is designed based on the idea of internal retransmission of messages after the timeouts and the return of the ACK to the sender machine kernel by the receiving machine kernel
- Kernel of the sending machine is responsible for retransmitting the message after waiting for the timeout period if no ACK is received from the receiving machine
- The sending machine, frees the sending process, only when the ACK is received
- The Timeout period is slightly more than round trip time for the message plus the avg. time required for executing the request
- Based on this idea, a four message reliable IPC protocol for client-server communication between two processes as shown

Four message reliable IPC protocol



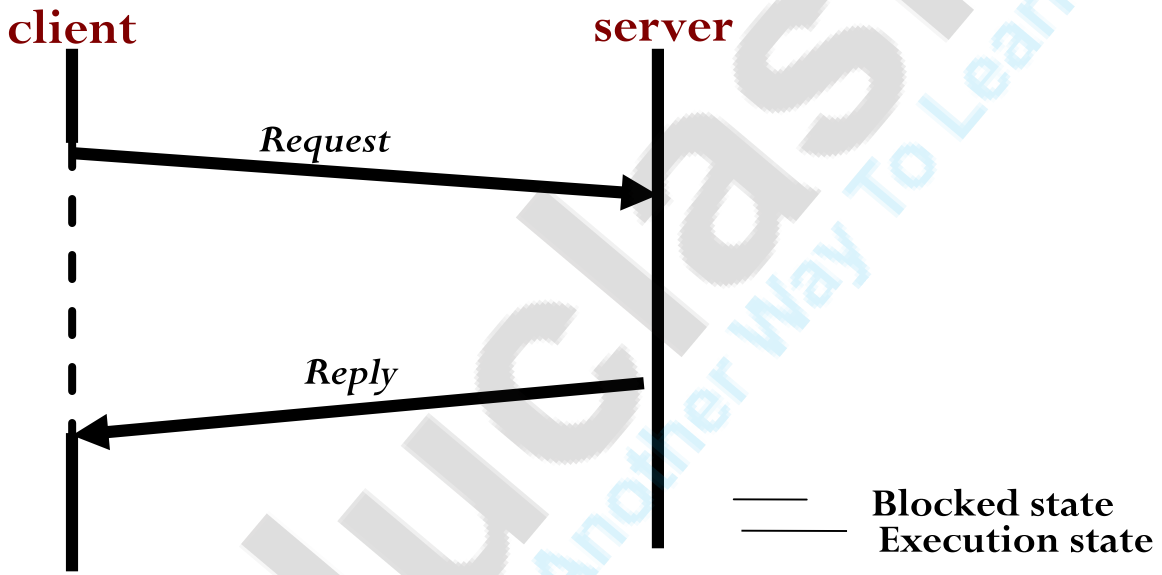
1. the client send a request message to the server
2. When the request message is received by the server's machine, the kernel sends an ACK to the sender machine. If the ACK is not received within time out, it resends the message
3. When the server finishes processing the client's request, it returns a reply message (containing the result of processing) to the client
4. When the reply message is received client machine sends an ACK

Three message reliable IPC protocol



- In a client-server communication, the result of the processed request is a sufficient ACK.
- Based on this 3-message reliable IPC protocol is as shown above
- The Server can send separate acknowledgement if processing of request is taking more time than timeout value.

Two message reliable IPC protocol



- To address long time needed for processing, and unnecessary low value of time out resulting in retransmission of the message and unnecessary network traffic the above two message protocol may be used

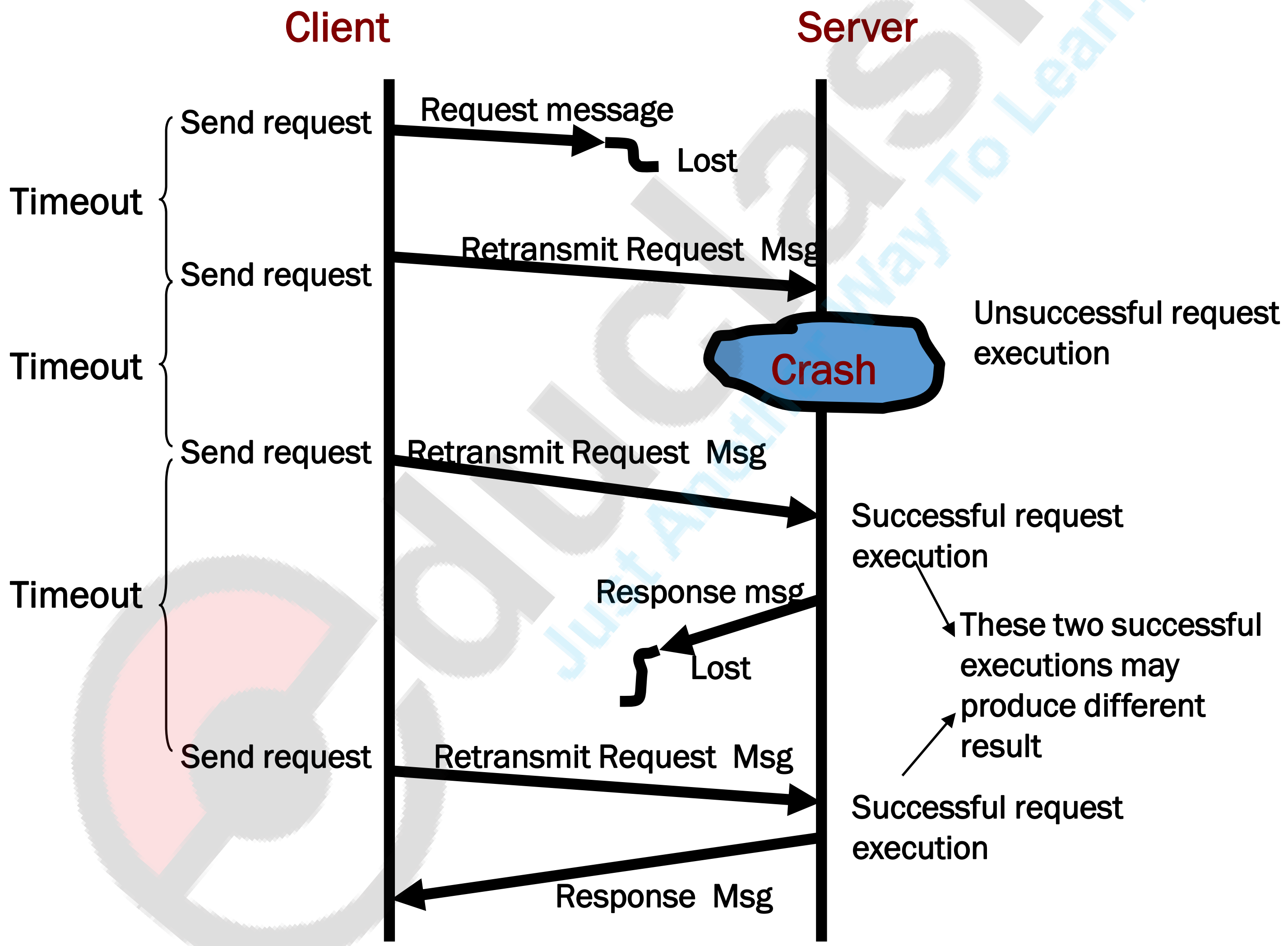
1. Client sends a request message to the server
2. Server starts a timer and if the processing is complete within time out the reply of the processing acts also as ACK, else server sends a ACK to the client

Fault Tolerant Communication

3. ACK from the client is not necessary as if reply not received as the client can repeat the request if the reply not received within time out

- Based on the two message protocol, an example of failure handling during communication between two processes is shown in the next slide
- The two message protocol is said to obey *at-least-once* semantics, which ensures that at least one execution of the receiver's operations has been performed(or perhaps more)
- It is more appropriate to call it as *last-one* semantics because the results of the last execution of the request are used by the sender

Fault Tolerant Communication



At – least-once semantics/ Last one semantics

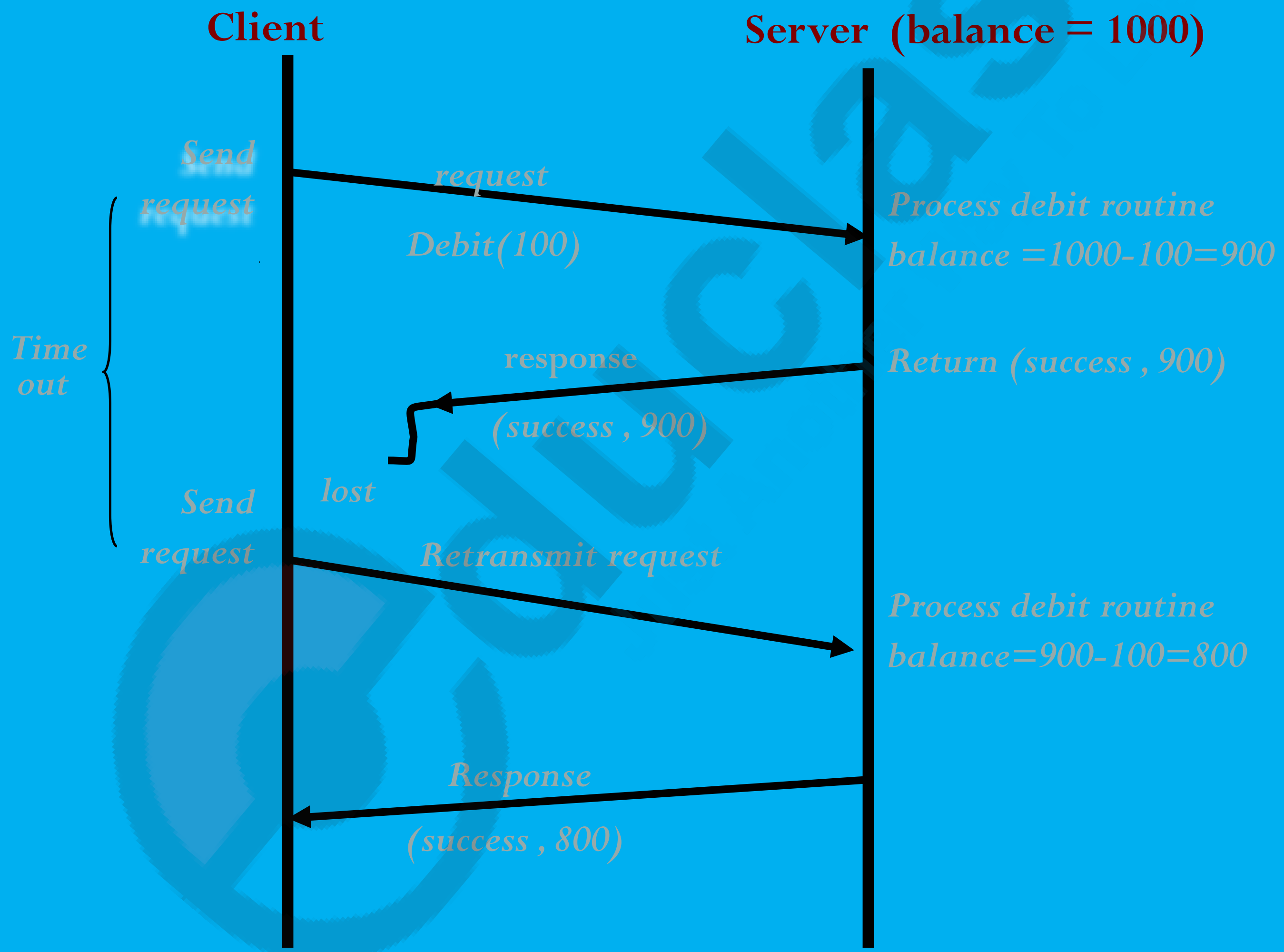
Idempotency

- Idempotency here basically means "Repeatability"
- An idempotent operation produces the same result without any side effect no matter how many times it is performed with the same arguments
- For example square root of a number *Getsqrt*(64) always return 8
- Operations that do not necessarily produce the same results when executed repeatedly are said to be nonidempotent
- For example consider the following routine that debits a certain amount from a bank account and returns the balance amount

```
debit(amount)
 if (balance ≥ amount)
    { balance = balance-amount;
      return ("Success", balance);}
else return ("Failure, balance);
end;
```

- The fig in the slide shows the sequence of operations for debit(100)

Idempotency



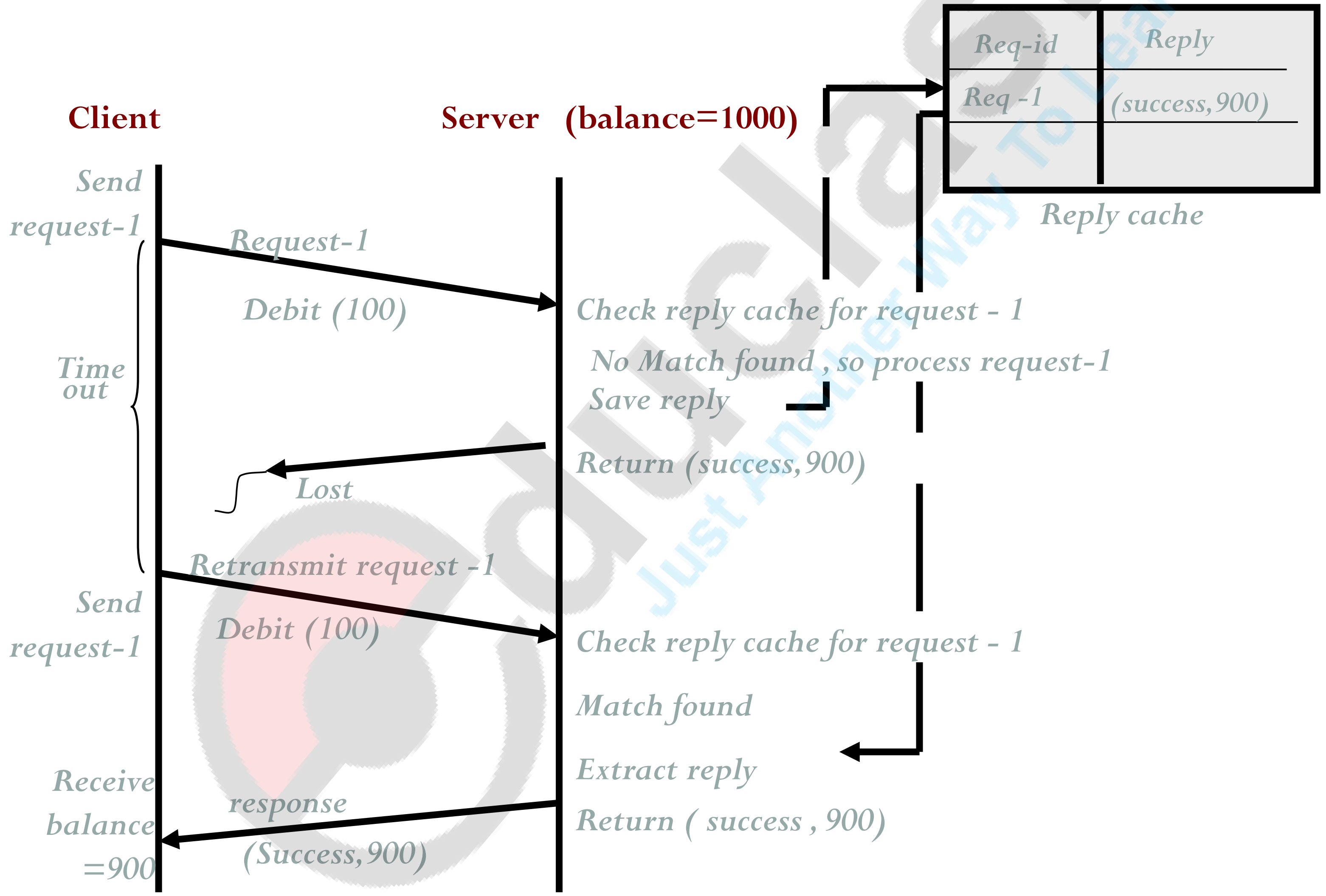
Handling Duplicate Request

- Using the timeout-based retransmission of request , the server may execute the same request message more than once
- If the execution is non-idempotent, its repeated execution will destroy the consistency of information
- Hence such "orphan" execution must be avoided
- This has lead to the *exactly–once* semantics, when used, it ensures that only one execution of server's operation is performed
- Use a unique identifier for every request that the client makes and to set up a reply cache in the kernel's address space on the server machine to cache replies

Handling Duplicate Request

- Then before forwarding request to the server for processing, the kernel of the processing machine checks to see if a reply already exists in the reply cache for the request
- If yes, it is duplicate request and hence, the previously computed result is extracted from the reply cache and a new response message is sent to the client
- Else the request is new one and sent to the server for processing

Handling Duplicate Request



- Ques. Which of the following operations are idempotent?
  1. Read_next_record(filename)
  2. Read_record(filename, record_no)
  3. Append_record(filename, record)
  4. Write_record(filename, after_record_n,record)
  5. Seek(filename, position)
  6. Add(integer1,integer2)
  7. Increment(variable_name)

- Ques. Which of the following operations are idempotent?
  1. Read_next_record(filename)
  2. Read_record(filename, record_no)
  3. Append_record(filename, record)
  4. Write_record(filename, after_record_n,record)
  5. Seek(filename, position)
  6. Add(integer1,integer2)
  7. Increment(variable_name)

Handling lost and out-of-sequence packets in multidatagram messages

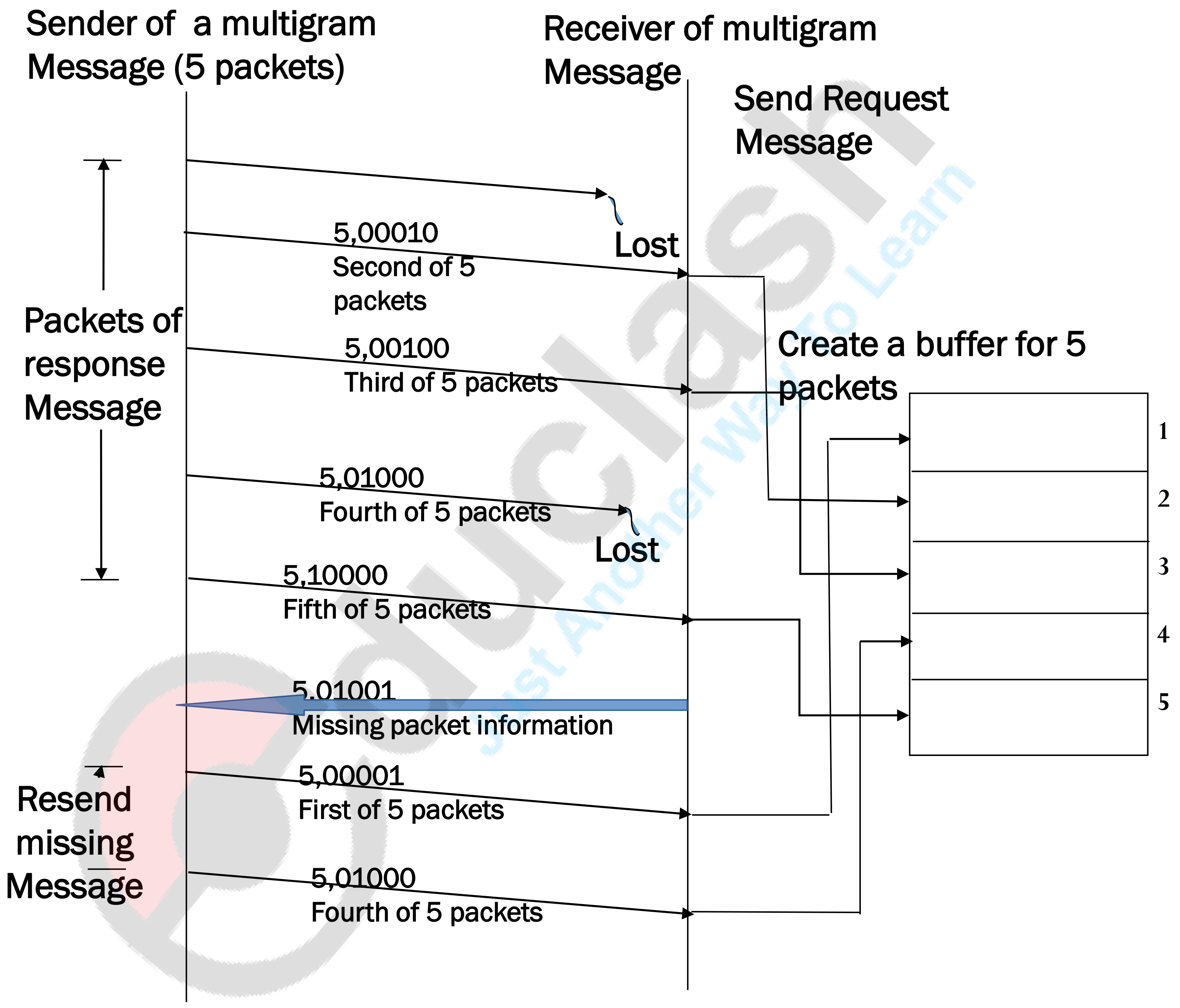
- In case of multidatagram messages, the logical transfer of a message consists of physical transfer of several packets
- Hence for completion of message transmission all the packets have to be received in the right order
- Hence reliable delivery of every packet is important
- Simplest way is to have ACK for each packet separately, which is called as Stop-and-wait protocol
    - This leads to excessive Communication Overhead
- To improve communication performance, a better approach is to have one ACK for all packets of multidatagram message (Blast protocol)

Handling lost and out-of-sequence packets in multidatagram messages

- Blast protocol: A node crash or communication failure can lead to
  - One or more lost Packets in communication
  - Packets are received out of sequence
- An efficient mechanism to cope with these problems is to use bitmap to identify the packets of message
  - Header has two extra fields- total no. of packets in the multidatagram message and position of this packet in complete message. e.g., (5, 00001, data) for 1st packet, (5, 00010, data) for 2nd and so on
  - First field helps the receiver to set aside sufficient buffer space
  - So even if out of sequence, data can be stored in buffer at the right position

Handling lost and out-of-sequence packets in multidatagram messages

- If at the time out all packets are not received, a bit map of unreceived packets is sent to sender. Using this information
  - Sender can do Selective resending of not received packets
    - If receiver sends (5,01001), sender sends back 1st & 4th packets again


Sender of a multigram Message (5 packets)
Receiver of multigram Message
Send Request Message
Lost
5,00010 Second of 5 packets
Packets of response Message
5,00100 Third of 5 packets
Create a buffer for 5 packets
1
2
5,01000 Fourth of 5 packets
Lost
3
4
5,10000 Fifth of 5 packets
5
5,01001 Missing packet information
Resend missing Message
5,00001 First of 5 packets
5,01000 Fourth of 5 packets

Group Communication

- The most common message based interaction is *one-to-one* communication also known as *point-to-point* or *unicast* communication
- Single process sends a message to a single receiver process
- However a good DCS often require group communication facility which are of three types
  - One to many
  - Many to one
  - Many to many
- Each of these have their own issues let us consider them

One to Many

- Multiple receivers for message sent by a single sender
- It is known as Multicast Communication
- A special case of multicast is Broadcast Communication where the message is sent to all the processors in the network
- Multicast/broadcast communication is very useful and has several practical applications
- To locate a processor providing particular class of service, one might broadcast across the network

Group Management

- In case of one to many communication, receiver processes form a group
- These group are of two types open and closed
- Open Group
  - Any process can send message to a group as a whole. e.g., Group of replicated servers
  - Outsider can send a message to all group members announcing its joining.
- Closed Group
  - Only members of a group can send message to the group. e.g. Collection of processors, Parallel processing a single application
  - Closed group also have to be open with respect to joining
- Whether to use an open group or closed group is application dependent

Group Management

- A good MPS should support both types of groups
- It should also provide flexibility to create and delete groups dynamically and allow a process to join and leave a group at any time
- Hence the MPS should have mechanism to manage the groups and their membership information
- A simple mechanism is Centralized group server process
  - All requests to create, delete a group, add or remove membership of a group are sent to this process
  - Hence maintenance of up to date group information is simple and straight forward
  - Centralized group server maintains a list of process identifiers of all processes for each group

Group Management

- It suffers from poor reliability & scalability, which is a common problem of any centralized service
- To some extent the problem can be solved by replicating the group server. This leads to the problem of data consistency

GROUP ADDRESSING

- Two-level naming scheme is normally used for group addressing
  - High level group name
    - ASCII string name independent of location of processes in group
    - Used by user applications
  - Low level group name
    - Depends to a large extent on underlying hardware (can be also a multicast address)

Group Addressing

- Multicast address / Broadcast address: a special network address to which multiple machines can listen
- Networks that do not have multicast facility may have broadcasting facility.
- A packet sent to a broadcast address is delivered to all the machines in the network
- Hence the broadcast address can also be used as a low level name for a group
- For group addressing multicasting is more efficient than broadcasting.

- If a network does not support either multicast or broadcast addressing then
  - One to one communication (Unicast) to implement group communication
    - Low level name :- List of machine identifiers of all machines belonging to group
    - Packets sent = no. of machines in group → Expensive

Unbuffered Multicast/ Buffered Multicast

- Multicast is asynchronous communication
  - Sending process can't wait for the readiness of all receivers
  - Sending process not aware of all receivers belonging to the group
- How a receiver treats a message depends upon whether the multicasting mechanism is buffered or unbuffered
- For an unbuffered multicast, if the receiving process is not in a ready state to receive the message, it is lost
- In case of buffered multicast it is buffered for the receiving process and hence will receive eventually
- Send to all semantics
  - Message sent to each process of multicast group and the message is buffered until it is accepted by all the process

Unbuffered Multicast/ Buffered Multicast

- Bulletin Board semantics
  - Message addressed to channel instead of being sent to every individual process of the multicast group
  - From a logical point of view the channel acts like bulletin board
  - Receiving process copies message from channel instead of removing it
  - Bulletin semantics is more flexible than send-to-all semantics
    - The relevance of a particular message to a particular receiver depends on its state i.e. If process is in idle state
    - Messages not accepted within a certain time after transmission may no longer be useful; their value may depend on the state of the sender
  - For example in a multicast message for idle server of particular function, only the idle machines make a receive request and others may be busy with other processing

Flexible Reliability in Multicast

- Different applications require different levels of reliability
- Hence multicast primitives normally provide flexibility for user definable reliability. e.g., Sender of a multicast message can specify no. of receivers from whom reply is expected
- 0-reliable: No response expected from any of the receivers
- 1-reliable: sender expects response from any of the receivers
- m out of n reliable: m out of n receivers responses are expected
- All reliable: Sender expects response from every one of the receivers
- **Atomic Multicast**
  - All - or - nothing property; i.e., when a message is sent to a group by atomic multicast, it is either received by all the processes that are members of the group or not received by any one

Atomic Multicast

- Atomic multicast is not always necessary. for example 0-, 1- and m out of n reliability, atomic multicast is not required.
- A simple way to implement atomic multicast is to multicast a message and asking for an ACK from each one of the group members
- After the timeout it retransmits to all those from whom no ACK
- The process repeated until all the ACKs are received
- Required for all - reliable semantics
- Involves repeated retransmissions by sender

- What if sender/ receiver crashes or goes down?
- Include message identifier & field to indicate atomic multicast
- Receiver also performs atomic multicast of message
- Very expensive

Group Communication Primitives

- Send( ): unicast
- send_group( ): multicast
  - Simplifies design & implementation of group communication
  - Indicates whether to use name server or group server
  - Can include extra parameter to specify degree of reliability (no of users from which reply is expected) or atomicity.

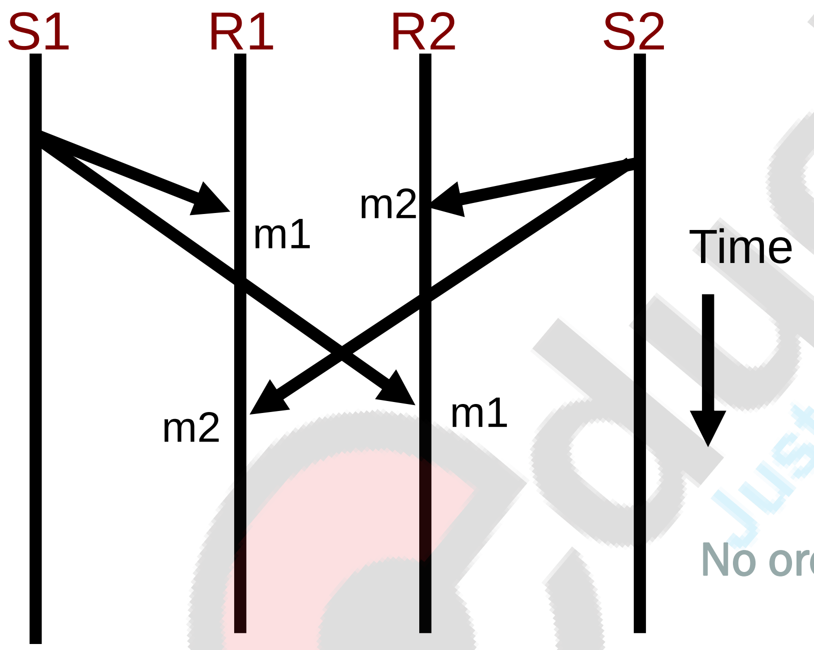
Many to one Communication

- Multiple senders – one receiver
- Selective receiver
  - Accepts from unique sender
- Non selective receiver
  - Accepts from any sender from a specified group
- E.g., producer-consumer process

Many-to-many Communication

- In this scheme, multiple senders send messages to multiple receivers
- One-to-many and many-to-one is implicit in this scheme
- Hence their issues are similar as described earlier
- One of the important aspect is Ordered message delivery
  - All messages are delivered to all receivers in an order acceptable to the application
    - Requires message sequencing
    - Out of order delivery is hardware dependent and depends on whether the network is a LAN or WAN
  - Hence ordered message delivery require special mechanisms

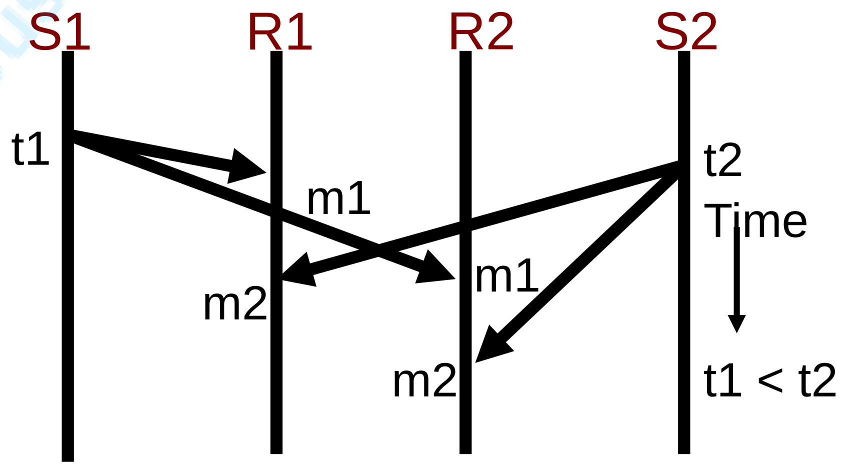
Many-to-many Communication



No ordering constraint for message delivery

Absolute Ordering

- Messages delivered to all receivers in the exact order in which they were sent
- Global timestamps are used as message identifiers and embedded in the message
- Kernel of each receiver saves all incoming messages meant for it in a separate queue
- Sliding window mechanism is used to deliver the message from the queue to receiver
- Fixed time intervals are selected for the window size and message falling with in the window are delivered to the receiver
- Absolute ordering semantics require globally synchronized clocks, which is not easy to implement

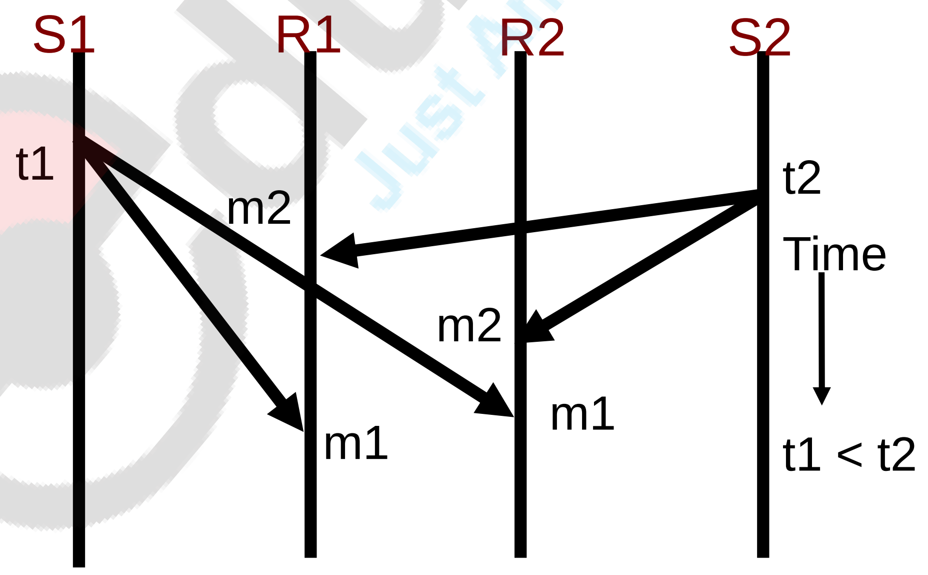


Consistent Ordering

- In most cases absolute ordering is not necessary
- It is enough to ensure all the receivers, receive the messages in the same order
- All messages are delivered to all receiver process in the same order, which may not the order in which messages were sent
- This is called consistent ordering and many systems support this
- One way of implementing consistent ordering semantics is to make the many to many scheme to appear as a combination of many-to one and one to many schemes. i.e., the kernel of the sending machines send messages to a single receiver (known as sequencer) that assigns a sequence no. to each message and then multicasts it. The kernel of the receiving machines store these messages in a separate queue and deliver the messages in the sequence no. ordering, if needed it will wait.

Consistent Ordering

- However this method is prone to single point failure and not reliable
- There is another stable system called ABCAST protocol where the sequence no. is assigned by distributed agreement among the group members



Consistent Ordering

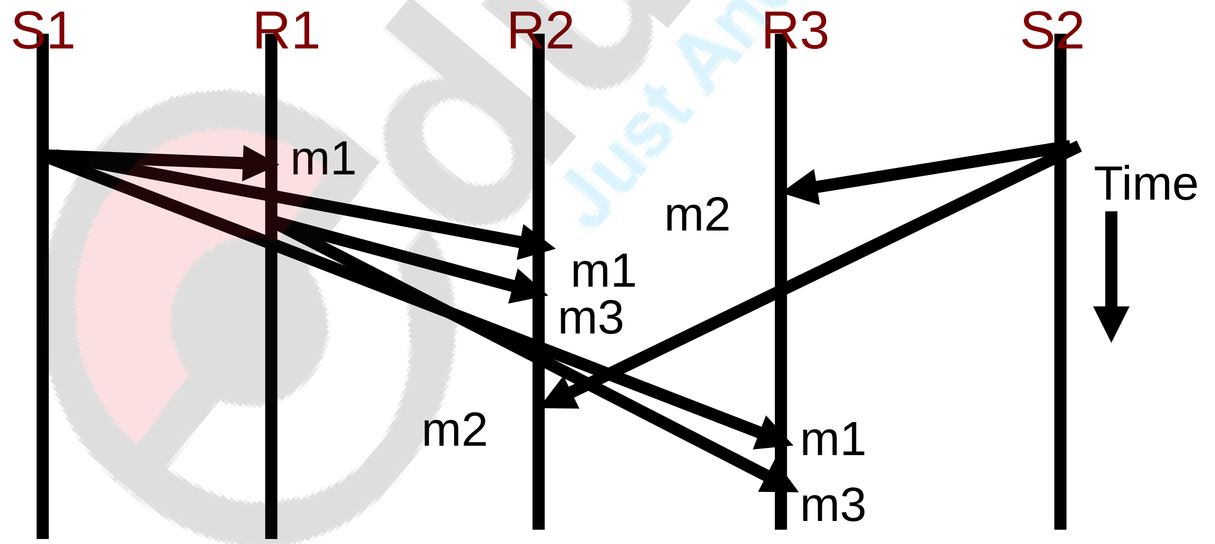
- Distributed algorithm (ABCAST)
  - Sender assigns temporary sequence no. larger than previous all nos., & sends to group.
  - Each member returns a proposed sequence no. to the sender. A member (i) calculates its proposed sequence number by $\mathbf{max(\ F_{max}, P_{max}) + 1 + i/N}$
  - When sender receives the proposed sequence nos. from all the members.
  - It selects largest sequence no. & sends to all members in a commit message. This chosen sequence no. is guaranteed to be unique
  - On receiving the commit message, each member attaches the final sequence number to the message
  - Committed messages are delivered to application programs in order of their final sequence nos.

Causal Ordering

- For some application even weaker semantics than consistent is acceptable
- Two message sending events casually related (any possibility of second message influenced by first one) then messages delivered in order to all receivers.
- The basic idea is that when it matters, messages are always delivered in the proper order, when it does not matter, delivery can be arbitrary
- Given the figure below Sender S_1sends a message m_1 to receivers R_1, R_2 and R_3 and sender S_2 sends message m_2 to receivers R_2 and R_3

Causal Ordering (Cont'd)

- On receiving m_1, R_1inspects and creates a new message m_3 and sends it to R_2 and R_3
- Note that event m_3 causally related the event of sending m_1
- Hence the two messages m_1 and m_3 should be received by R_2 and R_3 in that order
- Since m_2 is not related to m_1 or m_3 it can be delivered at any time to R_2 and R_3



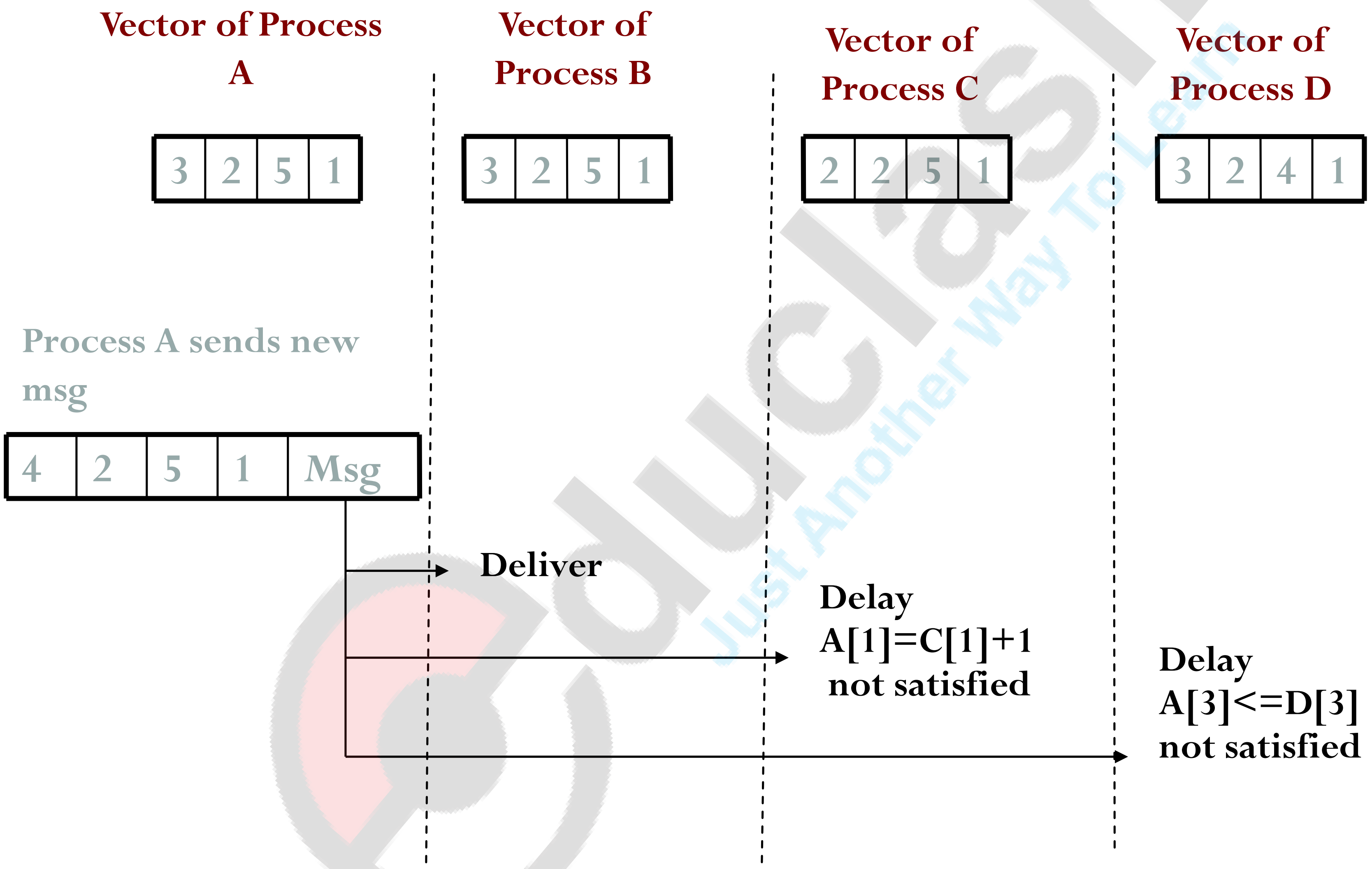
Causal Ordering (in ISIS)

- CBCAST algorithm
  - Consider a system with 4 processes A, B, C, & D
  - Status of their vector at any instant of time is as shown in the figure
  - This means that Until now, A has sent 3 messages, B→2, C→5 and D→1
  - Now A sends a new message and vector attached to it is (4, 2, 5, 1)
  - On arrival at a receiver, two conditions are tested let S be the vector of sender and R is the vector of receiver

S[i] = R[i] + 1 and S[j] <= R[j] for all j <> i and i is the sequence no of sender

  - Hence in above example message will be delivered to B, C will be delayed as A[i]=C[i] + 1 does not hold and for D, A[3] <= D[3] does not hold
  - First condition ensures that receiver has not missed any message from the sender.
  - Second condition ensures that sender has not received any message that receiver has not yet received.

CBCAST Protocol(Used in ISIS)



S[i]=R[i]+1 and S[j]<=R[j] for all j<>i

Case Study (IPC in Mach)

- *Mach* is a distributed system capable of functioning on heterogeneous systems.
- It incorporates multiprocessing support.
- System is highly flexible, i.e. able to run on systems with shared memory or no shared memory and single processors or multiple processors.

Port, socket and IP Address

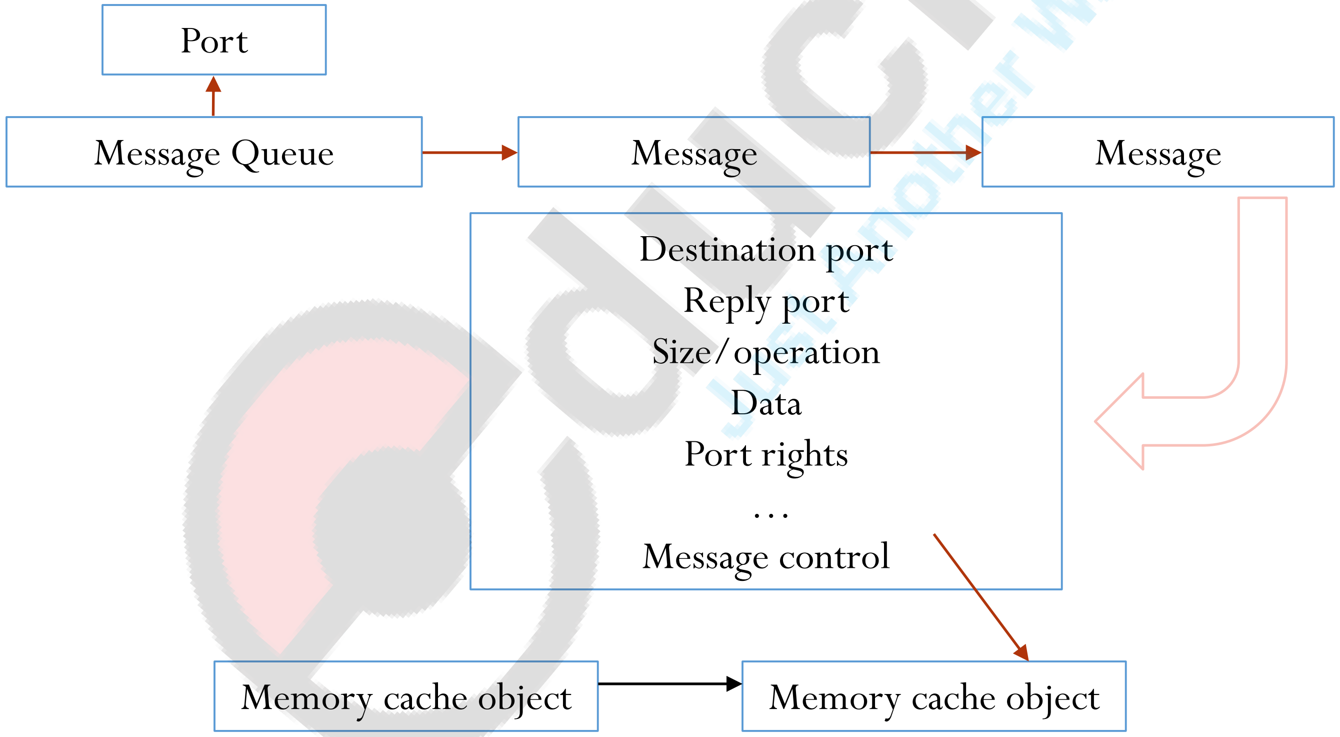
- Think of your machine as an apartment building:
  - A port is an apartment number.
  - A socket is the door of an apartment.
  - An IP address is the street address of the building.

Components of Mach IPC

- Two components of Mach IPC are ports and messages.
- Messages are sent to ports to initiate communication.
- Mach ensures security by assigning access rights to ports.
- A port is implemented as a protected and bounded queue.
- System calls that handles functionality of ports do the following:
  - Allocates port for specified task and give all access rights to the sender's task.
  - Deallocates a task's access rights to a port.
  - Get the current status of a task's port.
  - Creates a backup port.

Mach message format

- Message consists of fixed sized header and variable number of objects.
- Header contains the destination port name, the name of reply port to which return messages will be sent, length of message.



NetMsgServer

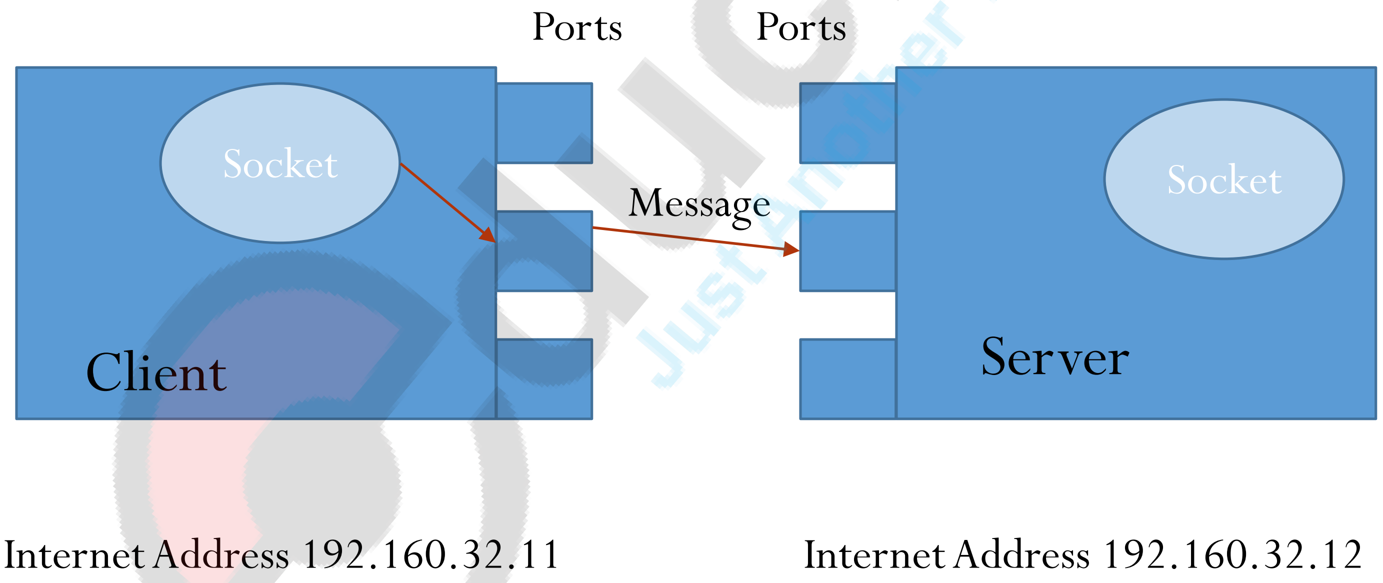
- *NetMsgServer* is used to:
  - Send a message between computers.
  - Provide location-transparent naming.
  - Extending IPC across multiple systems.
- It provides network-wide naming services.
- NetMsgServers maintain a distributed databases of port rights.
- Kernal uses NetMsgServer when passing messages:
  - Kernal transfers message to local NetMsgServer.
  - If service not available, message forwarded to other NetMsgServer by local NetMsgServer.

API for Internet Protocol

- *Socket Programming* used to implement interfaces for client server communication.
- Socket programming was first used by UNIX operating systems.
  - Sockets provides end to end communication.
  - Sockets are used in both UDP and TCP types of message passing.
  - A message is transmitted from a socket in one process to socket in another process.
  - Process binds its socket to a local port and to one of the host address.
  - Same socket can be used to send and receive messages.

UDP Communication

- For UDP datagram communication, a process must first create a socket and bind to the local port and to internet address of the local host.


Ports
Ports
Socket
Socket
Message
Client
Server
Internet Address 192.160.32.11
Internet Address 192.160.32.12

Norms for Datagram Communication

- Receiving process should specify the array size in which to receive the message.
- Sockets offer blocking receive and non-blocking send. Sometimes non-blocking receive can also be implemented.
- Receive method can receive a message from any source, it returns the internet address and the local port of sender allowing it to check origin of message. Sometimes connection is fixed also.
- Message communication should be reliable and valid. Error detection and error correction is there. Out of sequence delivery is taken care of.

TCP Datagram Communication

- TCP is connection oriented protocol.
- It ensures correct delivery of a long sequence of bytes as it establishes a bidirectional communication channel before starting message transfer.
- It maintains both the correctness and sequence of message delivery.
- It takes certain precautions while transferring data such as:
  - Sequencing: It maintains sequence with the help of sequence number embedded at the beginning of message.
  - Flow Control: it follows segment acknowledgements. When a segment is received , the sequence number is noted and receiver sends acknowledgement periodically along with the window size. Window size basically specifies the amount of data that can be sent before next acknowledgement.

- Retransmission and buffering: Retransmission is done in case receiver does not receive any packet and buffering is done to avoid speed mismatch between sender and receiver.

- Some services that use TCP protocol are
  - HTTP
  - FTP
  - Telnet
  - SMTP

University Questions

- How many types of reliabilities are there in group communication? Give two example of each.
- Multidatagram Messaging (short notes)
- How can you implement exactly once semantics.
- What are three different process address mechanisms. Give their relative advantages & disadvantages
- What are blocking & non blocking types of IPC. Give their relative advantages & disadvantages
- Write a short note on group communication in message passing
- What is ordered message delivery? Compare the various ordering semantics for message passing. Explain how each of these semantics is implemented?

- Give a mechanism for consistent ordering of messages in following cases:-
  - One-to-many communications
  - Many-to-one communications
  - Many-to-many communications