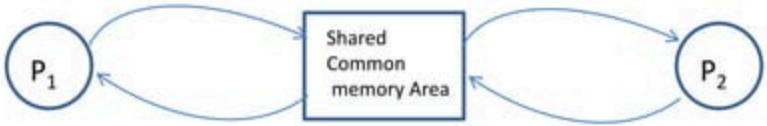
Chapter 3: Message Passing

Presented By
Dr.A.Ashok Kumar
Department Of Computer Science
Alagappa Govt. Arts College
Karaikudi - 630003

- A *process* is a program in execution
- *Resource manager process* to monitor the current status of usage of its local resources
- All resource managers communicate each other from time to time to dynamically balance the system load
- Therefore a DOS needs to provide *interprocess communication (IPC)* mechanism for communication activities
- IPC basically requires information sharing among two or more processes
- Two basic methods for information sharing
  - Original sharing, or shared-data approach
  - Copy sharing, or message-passing approach



- The *shared-data* paradigm gives the conceptual communication pattern

- In *message-passing approach*, the information to be shared is physically copied from the sender process address space to the address spaces of all receiver processes
- This done by transmitting the data in the form of messages
- A *message* is a block of information
- Communication processes interact directly with each other

- Distributed system communicate by exchanging messages
- Message passing is the basic IPC mechanism in distributed system
- Message-passing system is a subsystem of a DOS that provides a set of message-based IPC protocols
- It enables processes to communicate by simple communication primitives *send* and *receive*

Desirable Features of a Good Message-Passing system

1. Simplicity

- MPS should be simple and easy to use
- Construction of new applications and to communicate with existing ones by using the primitives provided by the MPS
- Different modules of a distributed application use simple primitives without bothering the system or network
- Use of clean and simple semantics of IPC protocols

2. Uniform Semantics

- Uses two type of communication
  - ✓ Local communication – the communicating process are on the same node
  - ✓ Remote communication – the communicating processes are on different nodes
- The semantics of remotes communication should be close as possible to those of local communications

3. Efficiency

- If the MPS is not efficient , IPC may become so expensive
- Application users try to avoid its use in their applications
- An IPC protocol of a MPS can be made efficient by reducing the number of message exchanges during communication
- Some optimizations are
  - ✓ Avoiding cost of establishing and terminating connections between the same pair of processes of every exchange
  - ✓ Minimizing the cost of maintaining connections
  - ✓ Piggybacking of acknowledgement

4. Relaiability

- A reliable IPC protocol can cope up with failure problems and guarantees the delivery of a message
- Failure due to node crash or communication link failure
- Handling of lost messages usually involves acknowledgements and retransmissions on the basis of timeouts
- Another issues related to reliability is duplicate messages
- Duplicate messages because of event of failures or timeouts
- A reliable IPC protocol is also capable of detecting and handling duplicates
- Use sequence number to avoid duplicate messages

5. Correctness

- IPC system has group communication
- One sender to multiple receiver, multiple sender to one receiver
- Correctness related to IPC protocols group communication

- Issues related to correctness is
  - Atomicity
    - Ensures that every message sent to a group of receivers will be delivered to either all of them or none of them
  - Ordered delivery
    - Ensures that messages arrive at all receivers in an order acceptable to the application
  - Survivability
    - Guarantees that messages will be delivered despite of partial failure of processes, machines, or communication links

6. Flexibility

- Not all applications require the same degree of reliability and correctness of the IPC protocols
- Many applications do not require atomicity or ordered delivery of messages
- The IPC primitives should be such that users have the flexibility to choose and specify the types and levels of reliability and correctness requirements of applications
- Flexibility permit control flow as synchronous and asynchronous send/receive

7. Security

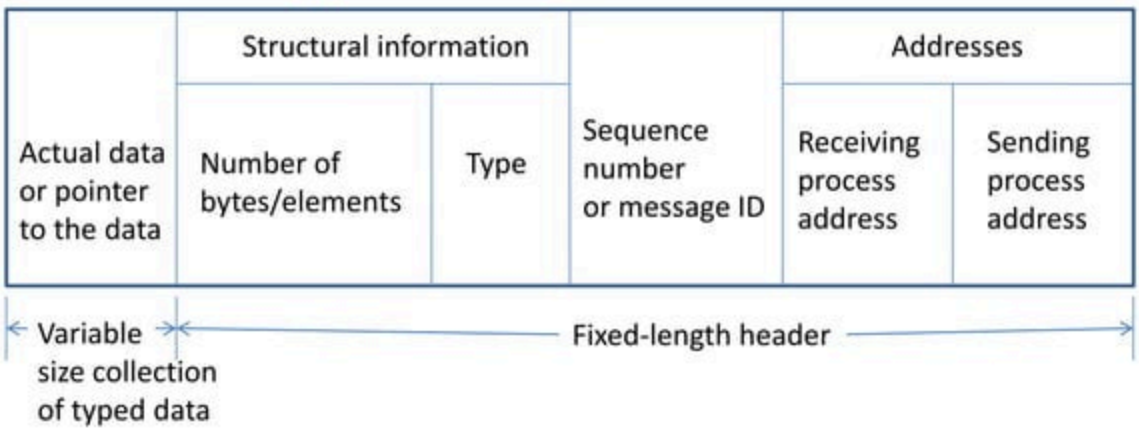
- A MPS be capable of providing a secure end-to-end communication
- A message in transit on the network should not be accessible to any user other than those to whom it is addressed and the sender
- Steps necessary for secure communication is
  - ✓ Authentication of the receiver(s) of a message by the sender
  - ✓ Authentication of the sender of a message by its receiver(s)
  - ✓ Encryption of a message before sending it over the network

8. Portability

- Two different aspects of portability
  - ✓ It should be easily construct new IPC facility on another system by reusing the basic design of existing MPS
  - ✓ Applications are also portable - heterogeneity must be considered while designing MPS

Issues in IPC by message passing

- 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-length header and a variable-size collection of typed data objects



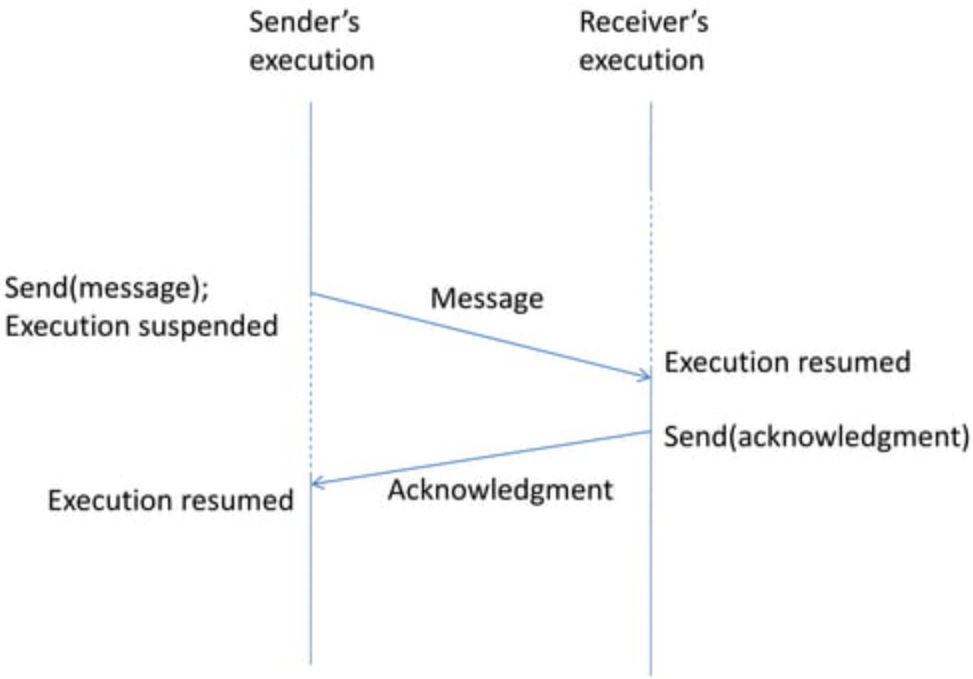
- In a message-oriented IPC protocol, the sending process determines the actual contents of a message
- The receiving process to convert the contents
- Special primitives are explicitly used for sending and receiving the messages
- Following issues to be discussed in the design of an IPC protocol
  - Who is the sender?
  - Who is the receiver?
  - Is there one receiver or many receivers?
  - Is the message guaranteed to have been accepted by its receiver(s)?
  - Does the sender need to wait for a reply?
  - What should be done in case of failure (crash or communication)?
  - What should be done if the receiver is not accept the message?
    - Will the message be discarded or stored in a buffer
    - In case of buffering, 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 messages?

Synchronization

- Major issue of communicating process is synchronization
- The semantics classified as *blocking* and *nonblocking* types
- *Nonblocking* semantics if it invocation does not block the execution of its invoker
- Otherwise a primitive is *blocking* (execution of invoker is blocked)
- Two types of semantics used for the *send* and *receive* primitives
- In case of *blocking send*, after the execution of send , the sending process is blocked until acknowledgement is received
- *Blocking receive*, after execution of receive statement , the receiving process is blocked until it receives message
- *Nonblocking send*, after sending process sending process is allowed to execute
- *Nonblocking receive*, the receiving process proceeds with its execution after execution the receive statement

- An important issue in a nonblocking receive primitive is how receiving process know that the message has arrived in the message buffer
- The following two methods used for this
  - ❖ Polling
    - ➢ A *test* primitive is allowed to the receiver to check the buffer status
    - ➢ Receiver periodically poll the kernel to check the buffer
  - ❖ Interrupt
    - ➢ When the message is filled in the buffer, *software interrupt* is used to notify the receiving process
    - ➢ This method permits the receiving process to continue without having unsucessful *test* requests
    - ➢ Its highly efficient and allows maximum parallelism
    - ➢ Drawback is user-level interrupts make programming difficult

- A variant of Nonblocking receive primitive is the *conditional receive primitive*
- It returns control immediately, either with a message or an indicator that no message
- Blocking send primitive uses the timeout values
- The value set by user or default value
- Timeout value used for blocking receive primitive to prevent the receiving process blocked indefinitely
- Both the send and receive primitives of a communication between two process use blocking semantics is said to be *synchronous*
- If its uses nonblocking primitives then communication is *asynchronous*
- Synchronous communication is simple and easy to implement
- Provide high reliability
- Drawbacks are
  - Limits the concurrency and is subject to communication deadlocks
  - Less flexible because sending process always has to wait for an acknowledgement, even it is nor required


Sender's execution
Receiver's execution
Send(message); Execution suspended
Message
Execution resumed
Send(acknowledgment)
Execution resumed
Acknowledgment

Buffering

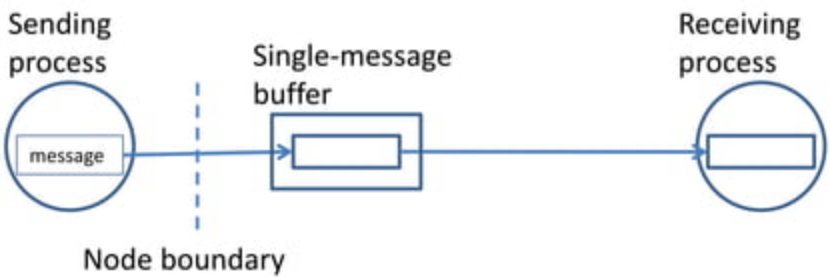
- Messages copying from the address space of the sending process to the address space of the receiving process
- If the receiving process is not ready to receive messages, then it should be save for later usage
- The message buffering is related to synchronization strategy
- The following are the buffering strategies
  1. Null buffer or no buffer
  2. Buffer with unbounded capacity
  3. Single-message buffer
  4. Finite-bound or multiple-message buffer

1. Null buffer (or no buffering)

- There is no place to temporarily store the message
- One of the following implementation strategies used
  - The message remains in sender address space and execution of *send* is delayed until the receiver executes *receive*
  - The message is simply discarded and the timeout mechanism is used to resend the message after a timeout period

2. Single-Message buffer

- A buffer capacity to store single message is used on the receiver's node
- An application module may have at most one message outstanding at a time
- Single-message buffer strategy is to keep the message ready for use at the location of the receiver
- The request message is buffered on the receiver's node if the receiver is not ready to receive the message
- The message buffer may either be located in the kernel's address space or in the receiver process's address space



3. Unbounded-capacity buffer

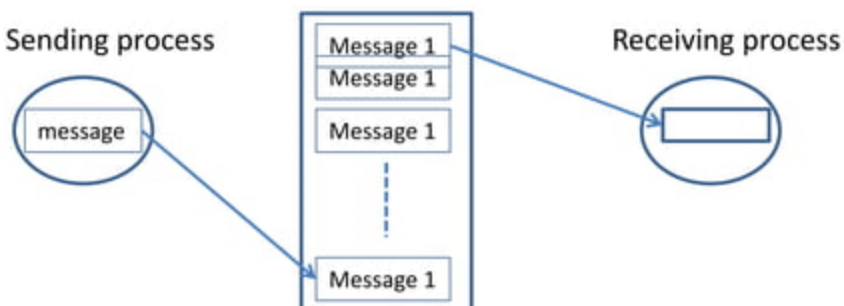
- A sender does not wait for the receiver to be ready
- An unbounded-capacity message buffer that can store all unreceived messages
- It assure that all the messages sent to the receiver will be delivered

4. Finite-bound (or multiple-message) buffer

- Asynchronous ode of communication use finite-bound buffers
- Need mechanism to handle the problem of buffer overflow
- Two ways to handle buffer overflow
  - Unsuccessful communication
    - Message transfers simply fail whenever there is no mode buffer space
    - The send normally returns an error message to the sending process
    - This method is less reliable

- Flow-controlled communication
  - The sender is blocked until the receiver accepts some messages
  - This method introduces a synchronization between sender and receiver
  - It result in unexpected deadlocks

- the amount of buffer space to be allocated depends on implementation
- A *create-buffer* system call is provided to the users
- The receiver mail box is located in the kernel address space or in the receiver process address space
- This buffering provides better concurrency and flexibility



Multiple-message buffer/mailbox/port

Multidatagram Messages

- All networks has upper bound of the size of data transmitted at a time
- This size is known as *Maximum Transfer Unit(MTU)* of a network
- Message size greater than MTU has fragmented in to multiples of the MTU
- Each fragment sent separately
- Each fragment is sent in a packet with control information and data
- Each packet is known as *datagram*
- Messages smaller than the MTU of the network can be sent in a single packet known as *single-datagram messages*
- Messages larger than the MTU of the network have to be fragmented and sent in multiple packets known as *multidatagram messages*

Encoding and Decoding of Message Data

- The structure of program objects should be preserved , while transmitting from the address of the sending process to receiving process
- Since both processes are on computers of different architectures it is difficult
- Because two reasons
  1. An absolute pointer value loses its meaning when transferred from one address space to another
  2. Different program objects occupy varying amount of storage space, ex. Long int, short int, var size character strings
- Due to this problem the program objects first converted to a stream form for transmission and placed into message buffer
- This conversion process on the sender side is known as *encoding* of a message data
- When received stream form converted to original program objects
- Known as *decoding*.

- Two representations used for the encoding and decoding

1. Tagged representation
   - The type of each program object along with its value is encoded in the message
   - The receiving process to check the type of each program object in the message
   - Program object is the self-describing nature of the coded data format
2. Untagged representation
   - The message data only contains program objects
   - No information is included in the message data to specify the type of each program object
   - Receiver process must have prior knowledge of how to decode

Process Addressing

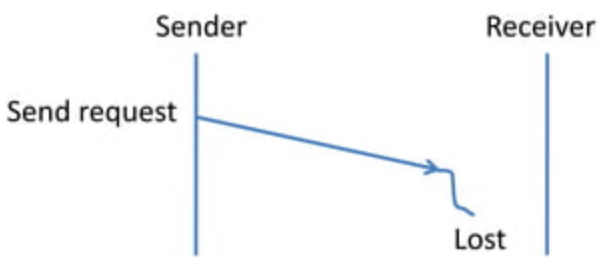
- MPS supports two types of process addressing
  1. Explicit Addressing
     - The process with which communication is desired is explicitly named as a parameter
     - *Send(process_id, message)* – send a message to the process identified by *process_id*
     - *Receive(process_id, message)* – receive a message from the process identified by *process_id*
  2. Implicit Addressing
     - A process willing to communicate does not explicitly name a process
     - *Send_any(service_id, message)*
     - *Receive_any(process_id, message)*
     - This type of primitive is useful in client-server communications when the client request from set of server provide services
     - This type of addressing is known as *functional addressing*

- The receiver is willing to accept a message from any sender
- Two types of process addressing used in communication primitives
  1. 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
     - The machine_id part is used by the sending machines machines kernel to send the message to the receiving process machine
  2. Another method process identified by a combination of three fields:*machine_id, local_id, and machine_id*
     - 1st field identifies the node on which the process is created
     - 2nd field is a local identifier generated by the node on which the process is created
     - 3rd field identifies the last known location (node) of the process
     - This method is known as link-based addressing

- Another method to achieve location transparency is name server
- A processer wants to send a message to another process specifies the high-level name of the process
- This name is mapped with *name server* to get its low-level names
- Using low-level name kernel sends the message to the proper name
- The sending kernel also caches the high-level name to low-level name receiving process
- For future use sending process need not contact name server again

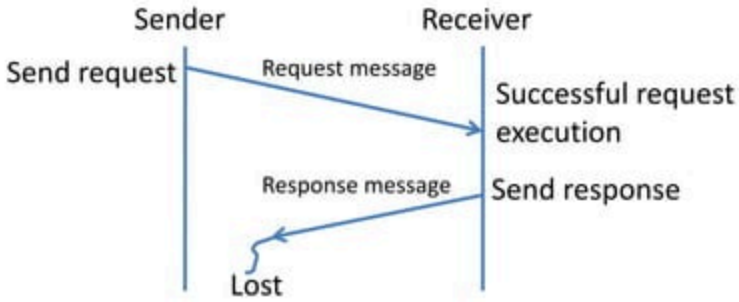
Failure Handling

- DS may suffer by partial failure like a node crash or a communication link failure
- During interprocess communication following failure problems will arise
  - Loss of request message
    - Due to the failure of communication link or receiver node is down at the time the request reaches there



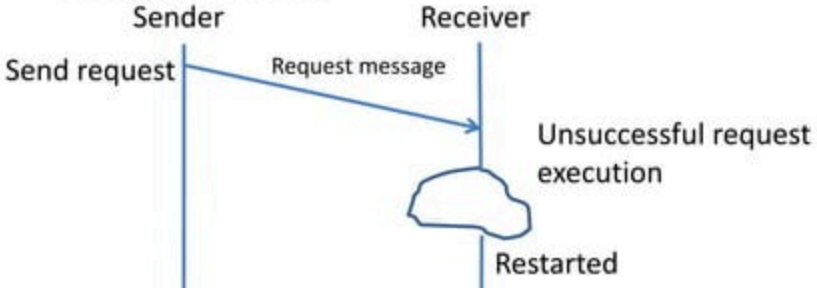
➢ Loss of response message

- Due to the communication link failure or the senders node down



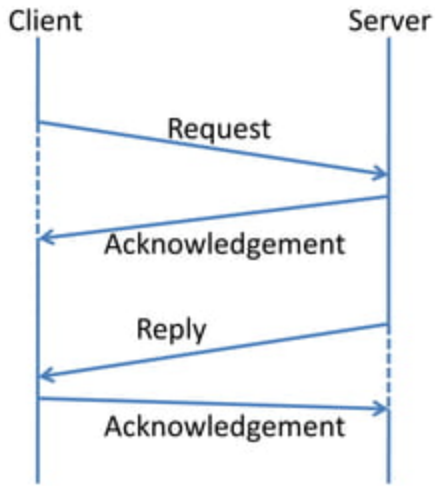
➢ Unsuccessfull execution of the request

- Due to the receivers node crashing while the request is being processed

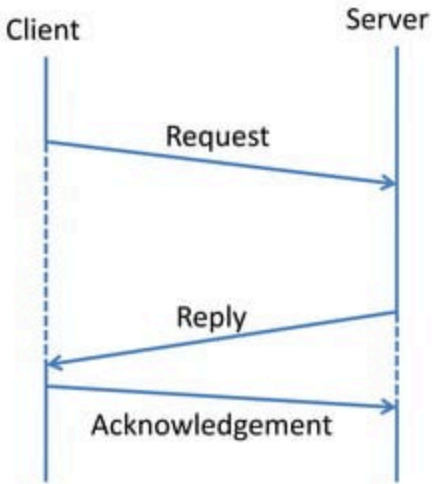


- The kernel of the sending machine is responsible for retransmitting the message after waiting for a timeout period
- If the no acknowledgement is received
- The kernel of the sending machine frees the sending process only when acknowledgement is received
- The timeout value is slightly more than the approximate round-trip time plus the average time required for executing request

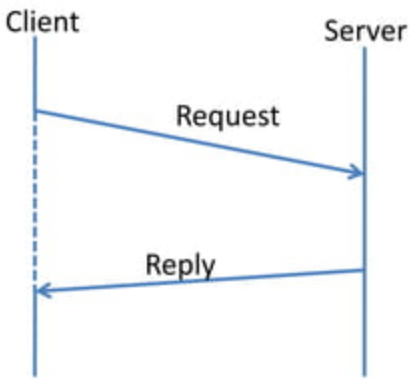
- Four-message IPC protocol for client server communication between two processes



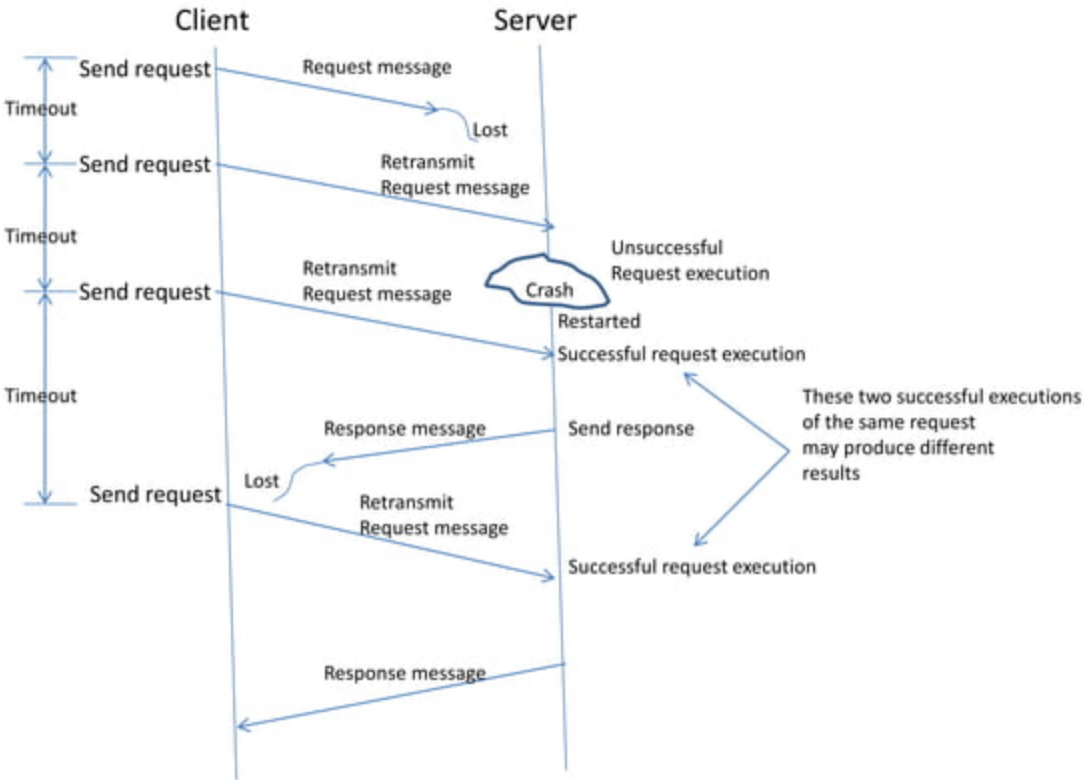
- In communication result of the processed request is sufficient acknowledgement
- Using this three-message IPC protocol defined



- In the above protocol a problem occurs if a request processing takes long time
- In this case the sender kernel timeouts and retransmit the message
- To avoid unnecessary retransmission use separate acknowledgement if processing takes long time
- Acknowledgement is not necessary because unnecessary retransmission of the messages and request processing
- Use two-message IPC protocol

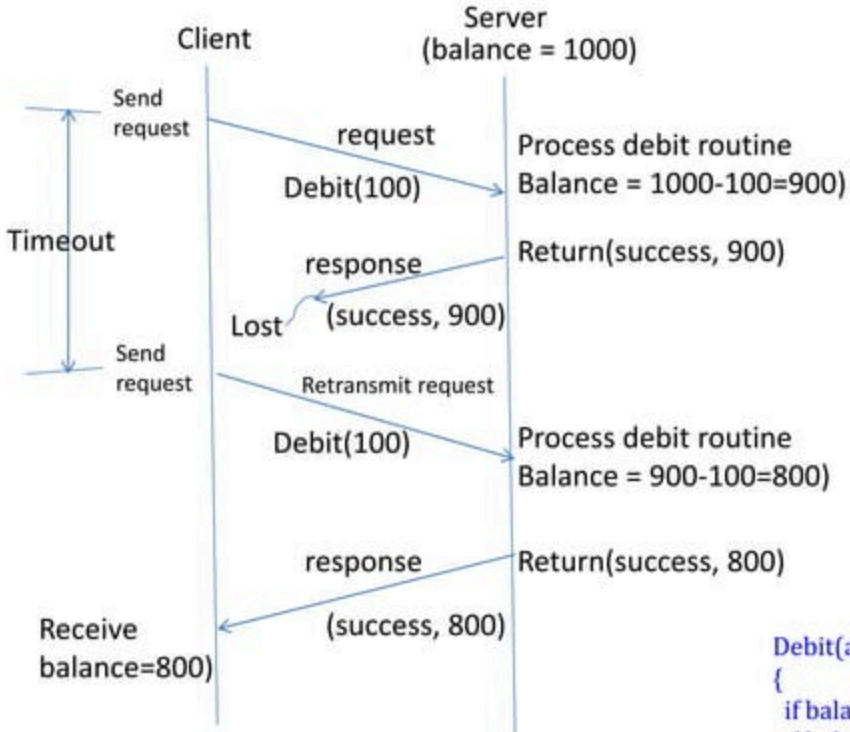


- Based on 2-msg IPC failure handling in communication



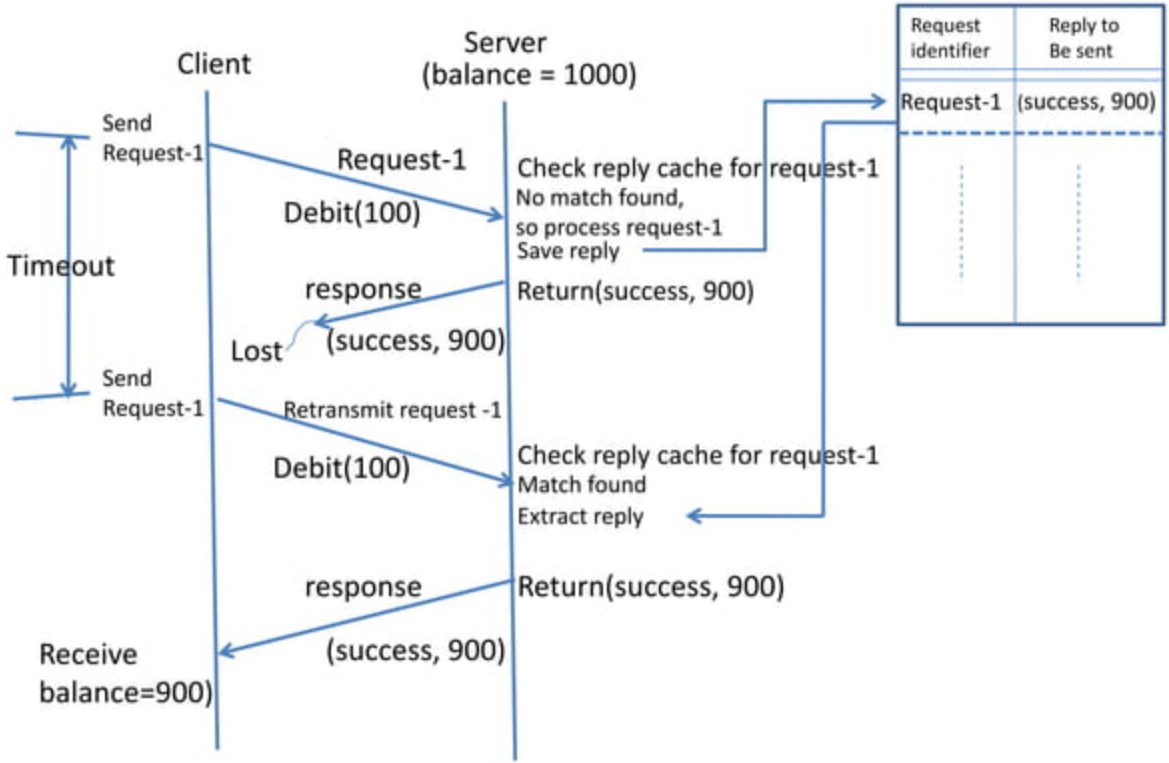
Idempotency and Handling of Duplicate Request Messages

- Idempotency means repeatability
- An idempotent operation produces the same results without any side effects
- Example for idempotent routine *GetSqrt(64)* produces the same value always 8.
- Operations which do not produce same results when executed repeatedly with same arguments are *nonidempotent*
- If the execution of the request is nonidempotent, then its repeated execution will destroy the consistency of information
- Such "*orphan*" executions must be avoided
- This orphan phenomenon use the exactly-once semantics, which ensures only one execution is performed by server
- It is difficult to implement
- Example of non idempotent execution




```
Debit(amount)
{
 if balance >= amount)
  { balance=balance-amount;
    return("success", balance);
  }
  else return("failure", balance);
}
```

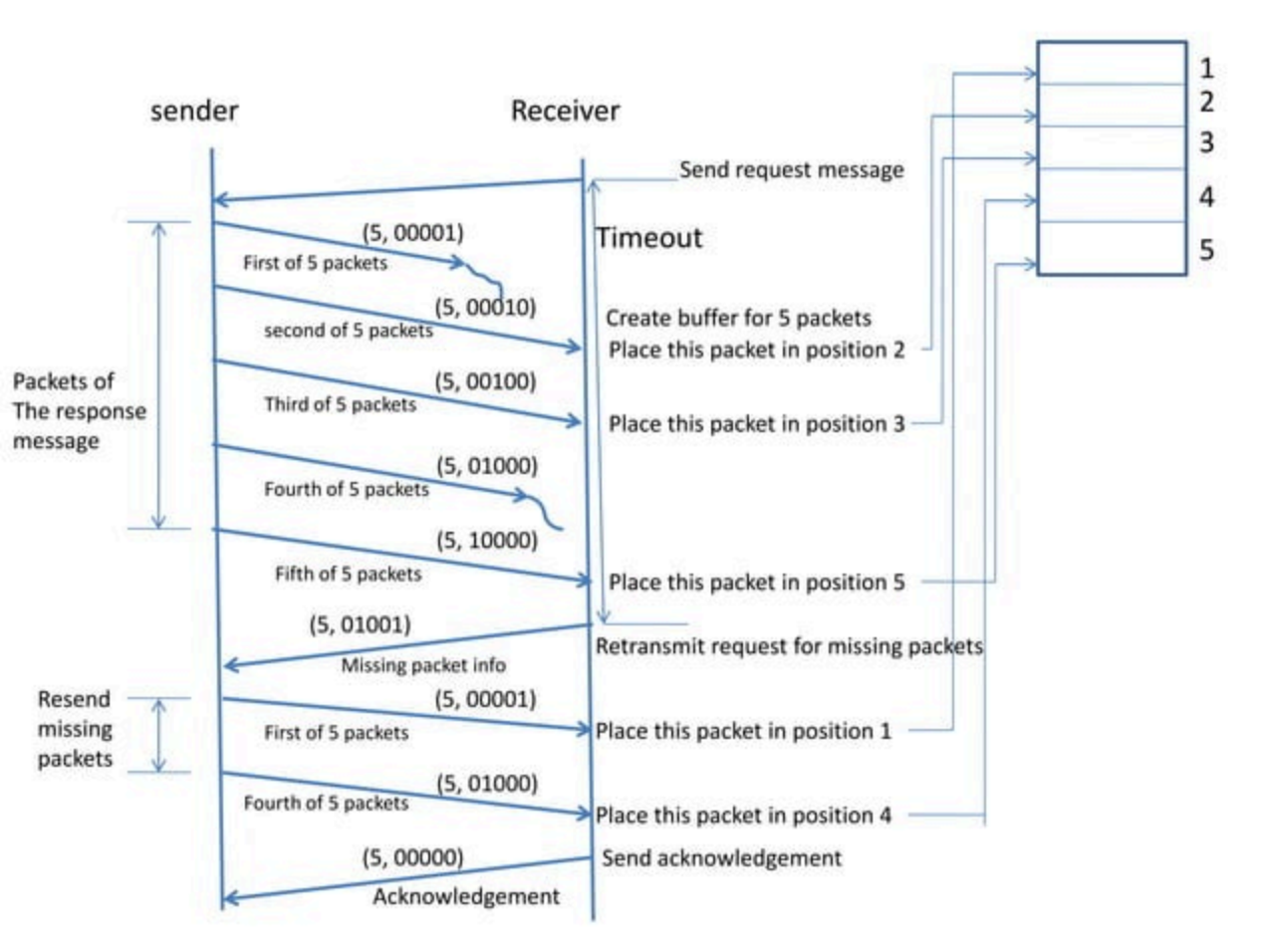
- Implementation of *exactly-once* semantics is to use a unique identifier for every request
- The client set up a reply cache in the kernels address space on the server machine to cache replies
- Before forwarding a request to a server for processing
  - Server checks to see if a reply already exists in the reply cache
  - If yes this is a duplicate request that has been already processed
  - Previously computer result is extracted and new response message sent to the client
  - Otherwise request is new one
  - After processing reply is stored along with the result of processing


Client
Server
(balance = 1000)
Request identifier
Reply to Be sent
Request-1
(success, 900)
Send Request-1
Request-1
Debit(100)
Check reply cache for request-1
No match found,
so process request-1
Save reply
Timeout
response
Return(success, 900)
Lost
(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)

Keeping Track of Lost and Out-of-Sequence Packets in Multigram Messages

- Message transmission is complete only all the packets of the message have been received
- Simple way to achieve is *stop-and-wait* protocol
- Separate acknowledgement leads to communication overhead
- Improve performance, better approach is to use a single acknowledgement packet for all the packets using *blast protocol*
- When this approach used it leads to the following problem due to node crash or link failure
  - One or more packets of the multidatagram message are lost in communication
  - The packets are received out of sequence by the receiver

- An efficient mechanism is to use a bitmap identify the packets of a message
- Header part contains two extra fields
  - One of which specifies total number of packets
  - Bitmap filed the specifies the position of this packet
- Receiving process set the buffer size according to the first field
- Packets stored in the corresponding buffer area
- After timeout, if all packet not received the bit map indicating unreceived packet send to sender
- This methods uses *selective repeat protocol*


sender
Receiver
Send request message
(5, 00001)
First of 5 packets
Timeout
(5, 00010)
second of 5 packets
Create buffer for 5 packets
Place this packet in position 2
Packets of
The response
message
(5, 00100)
Third of 5 packets
Place this packet in position 3
(5, 01000)
Fourth of 5 packets
(5, 10000)
Fifth of 5 packets
Place this packet in position 5
(5, 01001)
Retransmit request for missing packets
Missing packet info
Resend
missing
packets
(5, 00001)
First of 5 packets
Place this packet in position 1
(5, 01000)
Fourth of 5 packets
Place this packet in position 4
(5, 00000)
Send acknowledgement
Acknowledgement
1
2
3
4
5