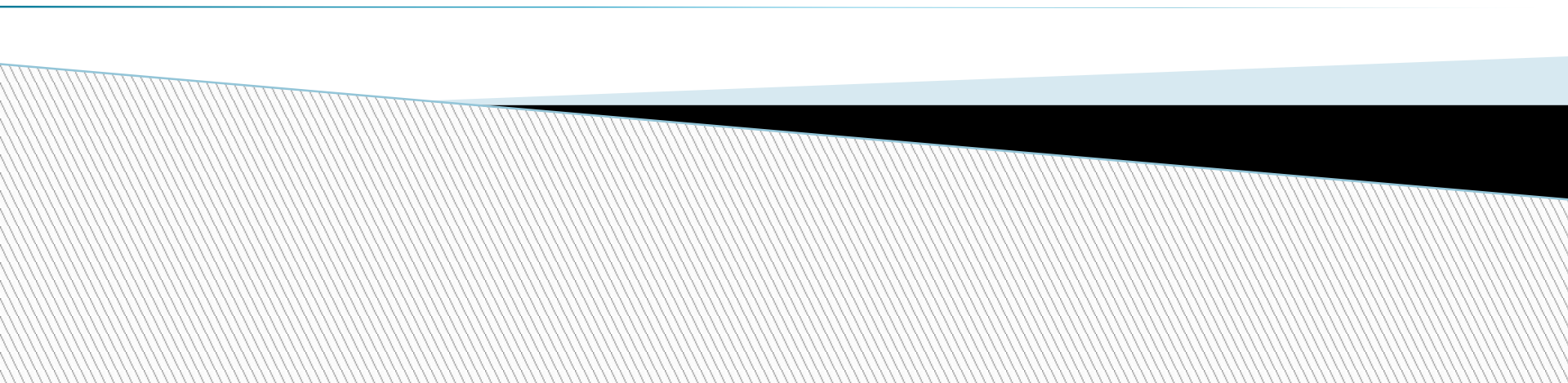


Message Passing



Buffering

? How do message transmitted from one process to another?

❑ Messages can be transmitted from one process to another by copying the body of the message from the address space of the sending process to the address space of the receiving process

? Is the receiving process is always ready to receive a message?

❑ In some cases, the receiving process may not be ready to receive a message transmitted to it but it wants the operating system to save that message for later reception.

❑ In these cases, the operating system will rely on the receiver having a buffer in which messages can be stored prior to the receiving process executing specific code to receive the message.

Buffering

[?] The synchronous and asynchronous modes of communication correspond respectively to the two extremes of buffering:

- ❑ a null buffer, or no buffering
- ❑ buffer with unbounded capacity.

[?] Other two commonly used buffering strategies are:

- ❑ single-message buffer
- ❑ finite-bound or multiple-message, buffers.

Buffering

Null buffer (No buffering)

In case of no buffering, there is no place to temporarily store the message.

- ❑ Hence one of the following implementation strategies may be used in case of no buffering.(How sender and receiver communicate ?)

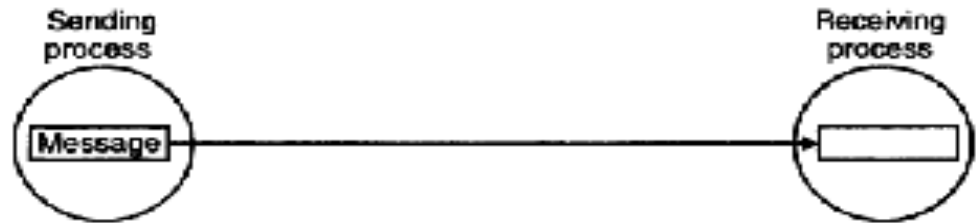
Case 1:

- ❑ The message remains in the sender process's address space and the execution of the send is delayed until the receiver executes the receive().
- ❑ When the receiver executes receive, an acknowledgment is sent to the sender's kernel saying that the sender can now send the message.
- ❑ On receiving the acknowledgment message, the sender is unblocked, causing the send() to be executed once again.
- ❑ This time, The message is successfully transferred from the sender's address space to the receiver's address space (since the receiver is waiting to receive the message).

Buffering

Null buffer (No buffering)

Case 2:



- ❑ After executing `send()`, the sender process waits for an acknowledgment from the receiver process.
- ❑ If no acknowledgment is received within the timeout period, it assumes that its message was discarded and tries again hoping that this time the receiver has already executed receive.
- ❑ **Note:**
 - The sender may have to try several times before succeeding.
 - The sender gives up after retrying for a predecided number of times.
- ❑ In the case of no buffering, the logical path of message transfer is directly from the sender's address space to the receiver's address space

Buffering

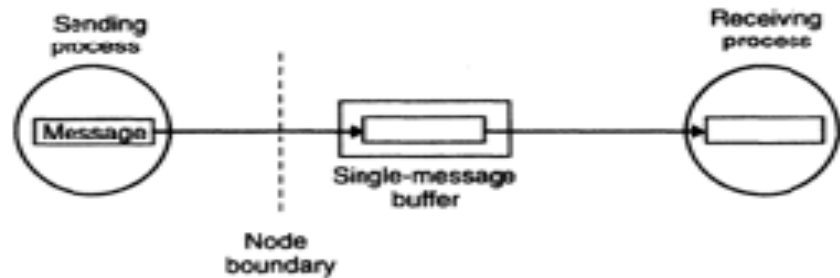
Null buffer (No buffering)

Important points to be noted:

- ❑ The null buffer strategy is generally not suitable for synchronous communication between two processes in a distributed system.
- ❑ This is because, if the receiver is not ready, a message has to be transferred two or more times, and the receiver of the message has to wait for the entire time taken to transfer the message across the network.
- ❑ In a distributed system, message transfer across the network may require significant time in some cases.
- ❑ Therefore, instead of using the null buffer strategy, synchronous communication mechanisms in network/distributed systems use a single-message buffer strategy.

Buffering

Single-Message buffer



- ❑ In this strategy, a buffer having a capacity to store a single message is used on the receiver's node.
- ❑ The main idea behind the single-message buffer strategy is to keep the message ready for use at the location of the receiver.
- ❑ In this method, 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**.
- ❑ The logical path of message transfer involves **two copy operations**.

Buffering

Unbounded-Capacity Buffer

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

Buffering

Finite-Bound (or Multiple.-Message) Buffer

- ❑ As Unbounded capacity of a buffer is practically impossible, in practice, **systems using asynchronous mode of communication use finite-bound buffers**
- ❑ A **strategy is needed** for handling the **problem of a possible buffer overflow** (the buffer has finite bound).
- ❑ The buffer overflow problem can be dealt with in one of the following two ways:
 - ❑ **Unsuccessful communication.**
 - ❑ **Flow-controlled communication.**
- ❑ *Unsuccessful communication:*
 - ❑ *Message transfers fail whenever there is **no more buffer space**.*
 - ❑ *The **send normally returns an error message** to the sending process, indicating that the **message could not be delivered** to the receiver **because the buffer is full**.*
 - ❑ *Unfortunately, the use of this method makes message passing **less reliable**.*

Buffering

Finite-Bound (or Multiple.-Message) Buffer

❑ *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 creating space in the buffer for new messages.*
- ❑ *This method introduces a synchronization between the sender and the receiver and **may result in unexpected deadlocks**.*

Note

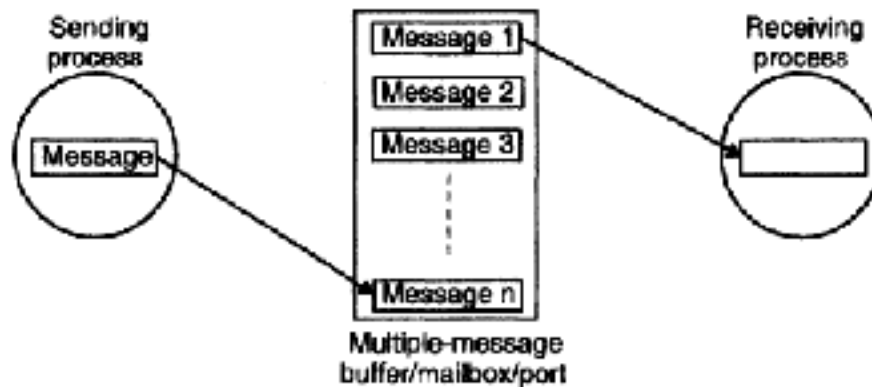
- ❑ *The **create buffer system call**, when executed by a receiver process, **creates a buffer** (sometimes called a **mailbox or port**) of a **size specified by the receiver**.*
- ❑ *The receiver's mailbox may be located either in the **kernel's address space** or in the **receiver process's address space**.*

Buffering

Finite-Bound (or Multiple.-Message) Buffer

Note

- ❑ In the case of *asynchronous send with bounded-buffer strategy*, the message is first copied
 - ❑ from the sending process's memory into the receiving process's mailbox
 - ❑ then copied from the mailbox to the receiver's memory when the receiver calls for the message.



PROCESS ADDRESSING

- ❑ Important issue in message-based communication is addressing (or naming) of the parties involved in an interaction.
- ❑ A message-passing system usually supports two types of process addressing
 - ❑ Explicit addressing
 - ❑ Implicit addressing

Explicit addressing.

- ❑ The process with which communication is desired is explicitly named as a parameter in the communication primitive used.
- ❑ Primitive used in Explicit addressing are
 - ❑ `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`”.

PROCESS ADDRESSING

Implicit addressing.

- ❑ A process willing to communicate does not explicitly name a process for communication.
- ❑ Primitive used in Implicit addressing are
 - ❑ **Send_any(service_id, message)**: Send a message to any process that provides the service of type “service_id”.
 - ❑ The sender names a service instead of a process.
 - ❑ This type of primitive is useful in client-server communications when the client is not concerned with which particular server out of a set of servers providing the service.
 - ❑ **receive_any(process_id, message)**: Receive a message from any process and return the process identifier (“process_id”) of the process from which the message was received.
 - ❑ The receiver is willing to accept a message from any sender.
 - ❑ This type of primitive is useful in client-server communications when the server is meant to service requests of all clients that are authorized to use its service.

PROCESS ADDRESSING

[?] Methods to identify a process

1. By a combination of

machine_id and local_id, such as machine_id@local_id.

Local id – Process identifier or port identifier – uniquely identifies a process on a machine.

Pros : No need of global coordinator

Cons: Does not support Process migration

2. By a combination of the following three fields:

machineld, local_id, and machineid.

Machine id part is used by sending machine kernel to send msg to receiving process machine.

Used by the kernel of receiver process to forward the msg to the process for which its intended

PROCESS ADDRESSING

- ? The first field identifies the node on which the process is created
- ? The second field is a local identifier generated by the node on which the process is created
- ? The third field identifies the last known location (node) of the process
- ? Link based process addressing.

Machine_id@local_id@machine_id

Drawback

- ? The overhead of locating a process may be large if the process has migrated several times.
- ? It may not be possible to locate a process if an intermediate node is down
- ? Location transparency

PROCESS ADDRESSING

❓ To achieve location transparency in process addressing, two level naming scheme for processes is used.

❓ Each process has two identifiers

high level name – Machine independent (ASCII)

Low level name- Machine dependent (machine_id@local_id)

Sender specifies high level name of receiving process . Sender kernel contacts name server to get low level name. using this kernel sends to proper machine. Receiver kernel delivers to receiving process

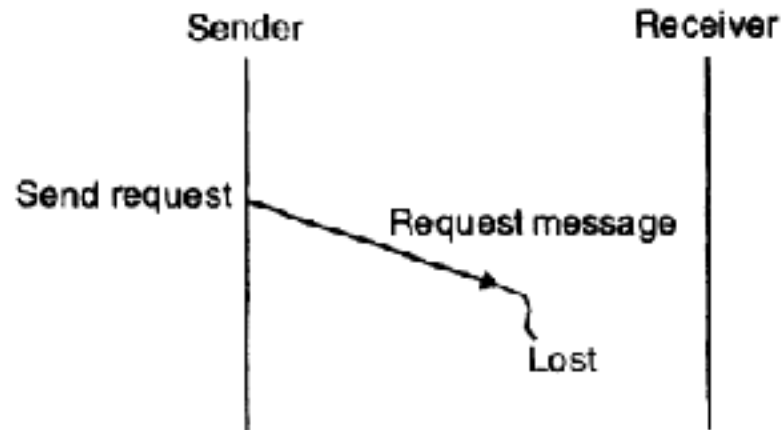
Caching is used. (high level name- low level name)

Pros – Suitable for functional addressing –high level name identifies service and name server maps to one or more process that provides service

Cons – Poor reliability and Scalability because name server – centralized. To overcome replicate – Increases overhead.

FAILURE HANDLING

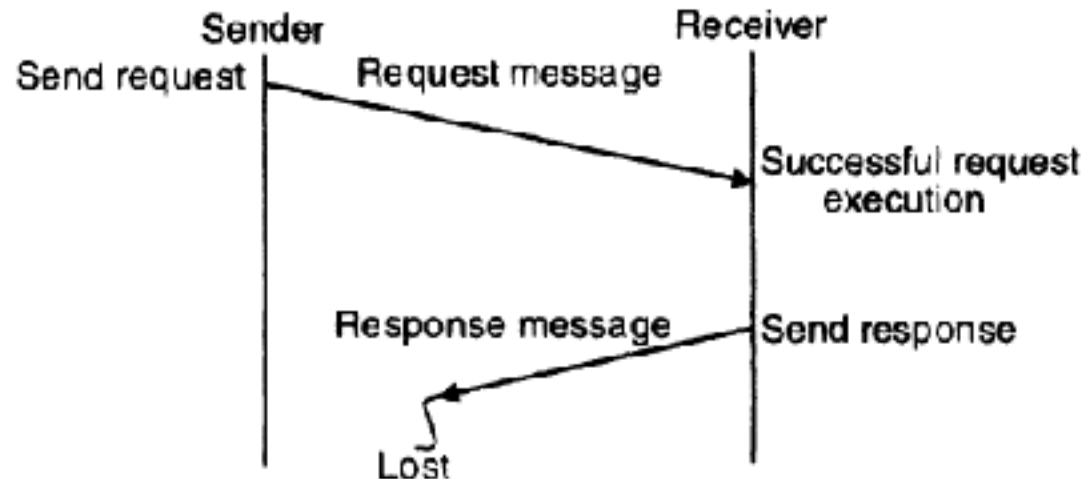
- ❑ Distributed system may be prone to partial failures such as a node crash or a communication link failure.
- ❑ During interprocess communication, such failures may lead to the following problems:
 - ❑ **Loss of request message:** This may happen either due to the **failure of communication link** between the sender and receiver or because the **receiver's node is down** at the time the request message reaches there



[?] During interprocess communication, such failures may lead to the following problems:

FAILURE HANDLING

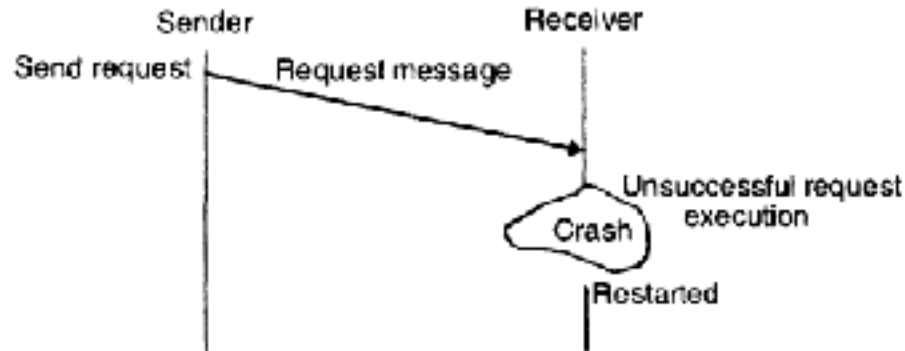
- ❑ **Loss of response message.** This may happen either due to the **failure of communication link** between the sender and receiver or because the **sender's node is down** at the time the response message reaches there.



FAILURE HANDLING

[?] During interprocess communication, such failures may lead to the following problems:

- ❑ **Unsuccessful execution of the request:** This happens due to the receiver's node crashing while the request is being processed.



[?] To cope with these problems, a reliable IPC protocol of a message-passing system is normally designed based on the idea of:

- ❑ **Internal retransmissions of messages after timeouts and the return of an acknowledgment message to the sending machine's kernel by the receiving machine's kernel.**

CLIENT

FOUR WAY PROTOCOL

SERVER

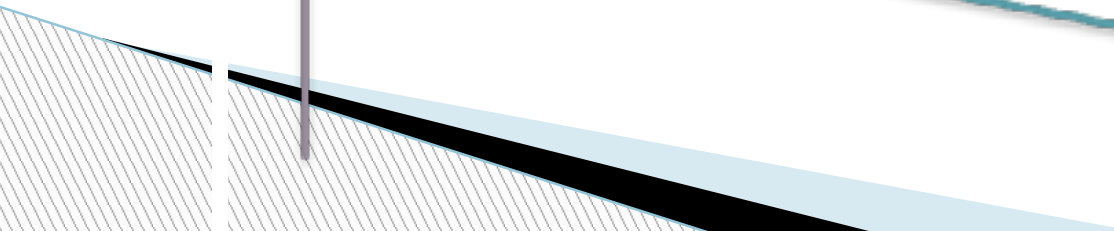
REQUEST

REQ
SUCESSS
SEND ACK

ACK

REPLY (result)

ACK



CLIENT

THREE WAY PROTOCOL

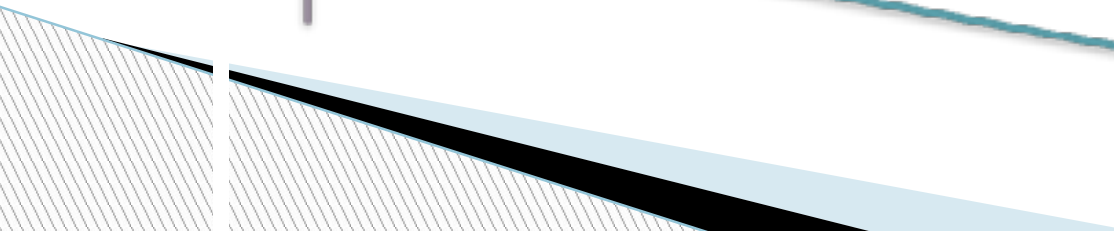
SERVER

REQUEST

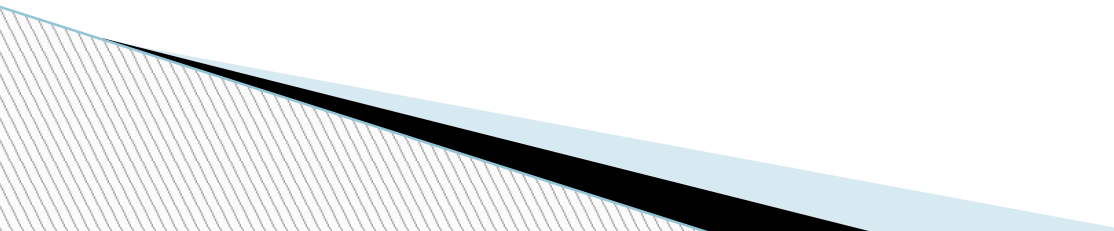
REQ
SUCESSSS
SEND
ACK

REPLY

ACK



Problems

- Request processing takes a long time.
 - Req msg- lost – retransmitted only after timeout (large value)
 - Timeout value is not set properly (large value) – unnecessary retransmission
- 

CLIENT

TWO WAY PROTOCOL

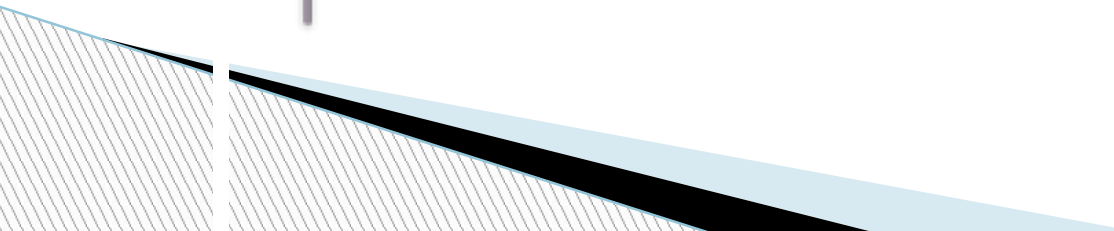
SERVER

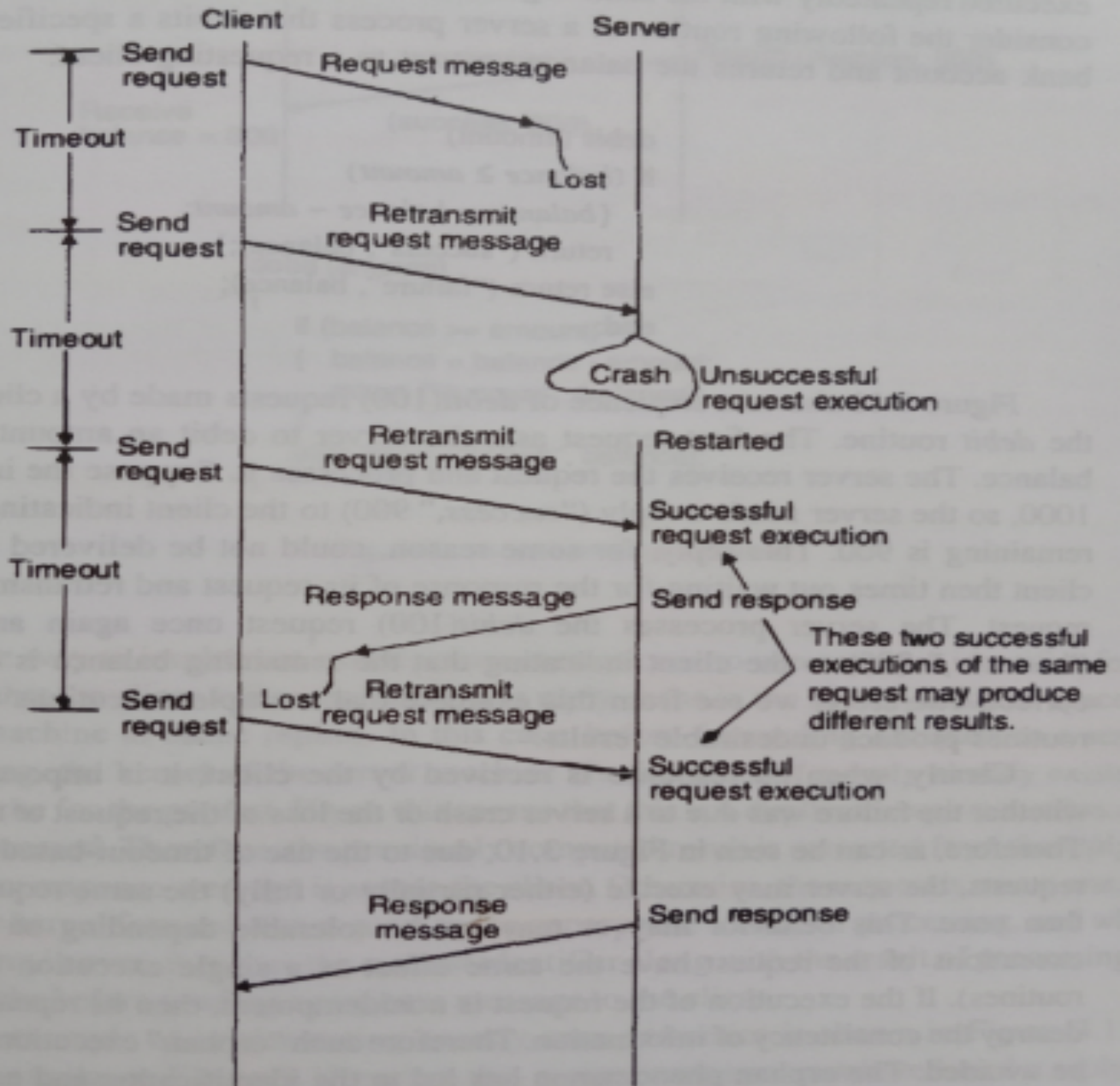
REQUEST

REQ
SUCESSSS
SEND ACK

REPLY

TIMER





Idempotency and Handling of Duplicate Request Messages

Repeatability

produces the same results with same arguments no matter how many times it is performed.

Eg. Debit(amount)

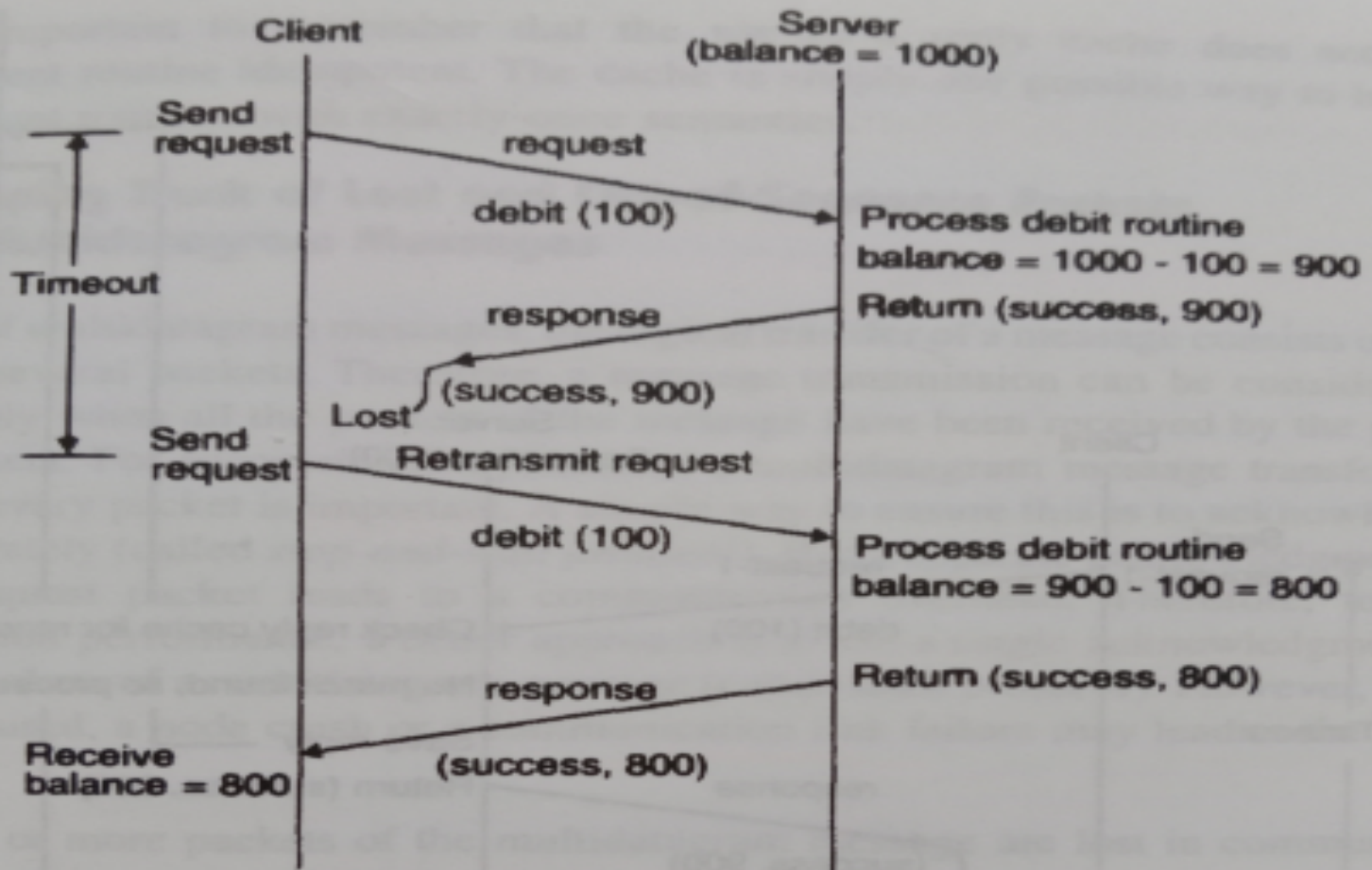
If(balance >= amount)

Balance = balance - amount;

Return("Success", balance);

Else Return("failure",
balance);

End;



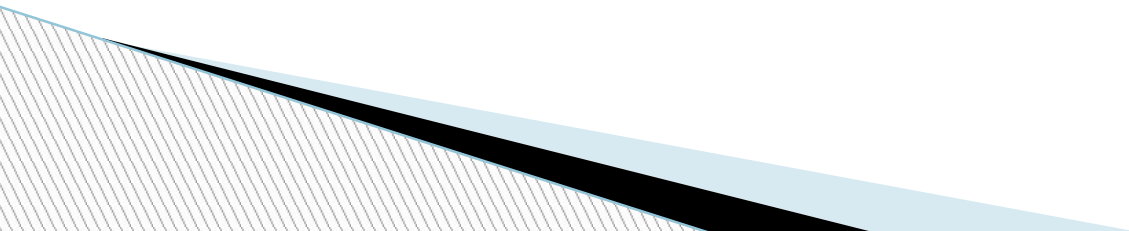
```

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

Fig. 3.11 A nonidempotent routine.

Exactly – once semantics

Can be implemented by using a unique identifier for every client request and set up a reply cache in the kernel address space on server machine



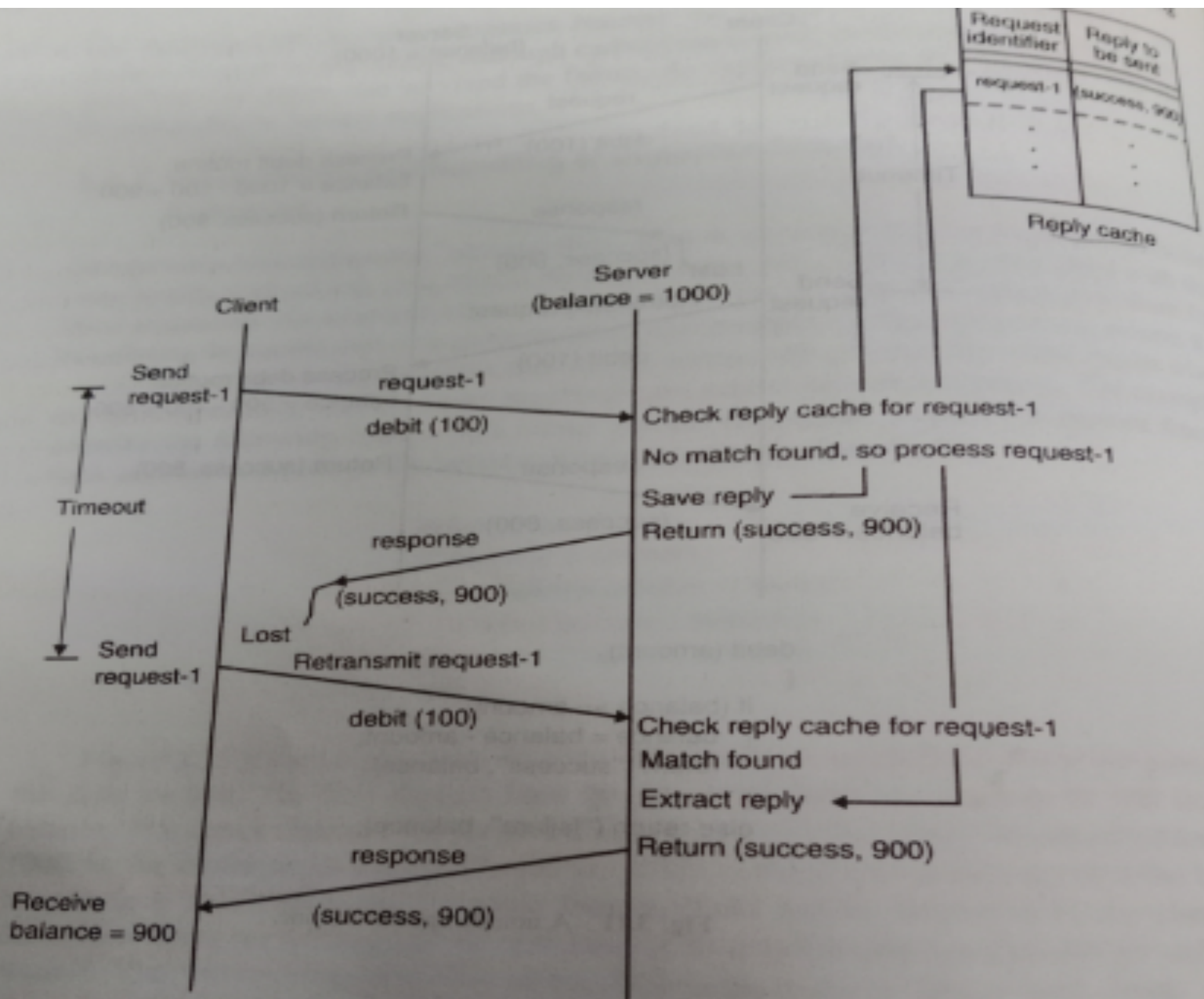


Fig. 3.12 An example of exactly-once semantics using request identifiers and reply cache.

lost and Out-of-Sequence Packets

For successful completion of a multidatagram message transfer, reliable delivery of every packet is important.

Reliability

- Stop and Wait Protocol
- Called Blast Protocol

link failure leads to

- One or more packet Loss
- Packets are received Out of sequence



Ack for each packet

Ack for all

lost and Out-of-Sequence Packets

Packet Header Part two extra fields

- First field- Total no of packets in multidatagram msg
- Second - bitmap field that specifies the Position of this packet in complete message.
- After timeout not all packets are received, bitmap indicating unreceived packets will be sent to sender .Sender retransmits only the packets that is not yet received.

Selective Repeat



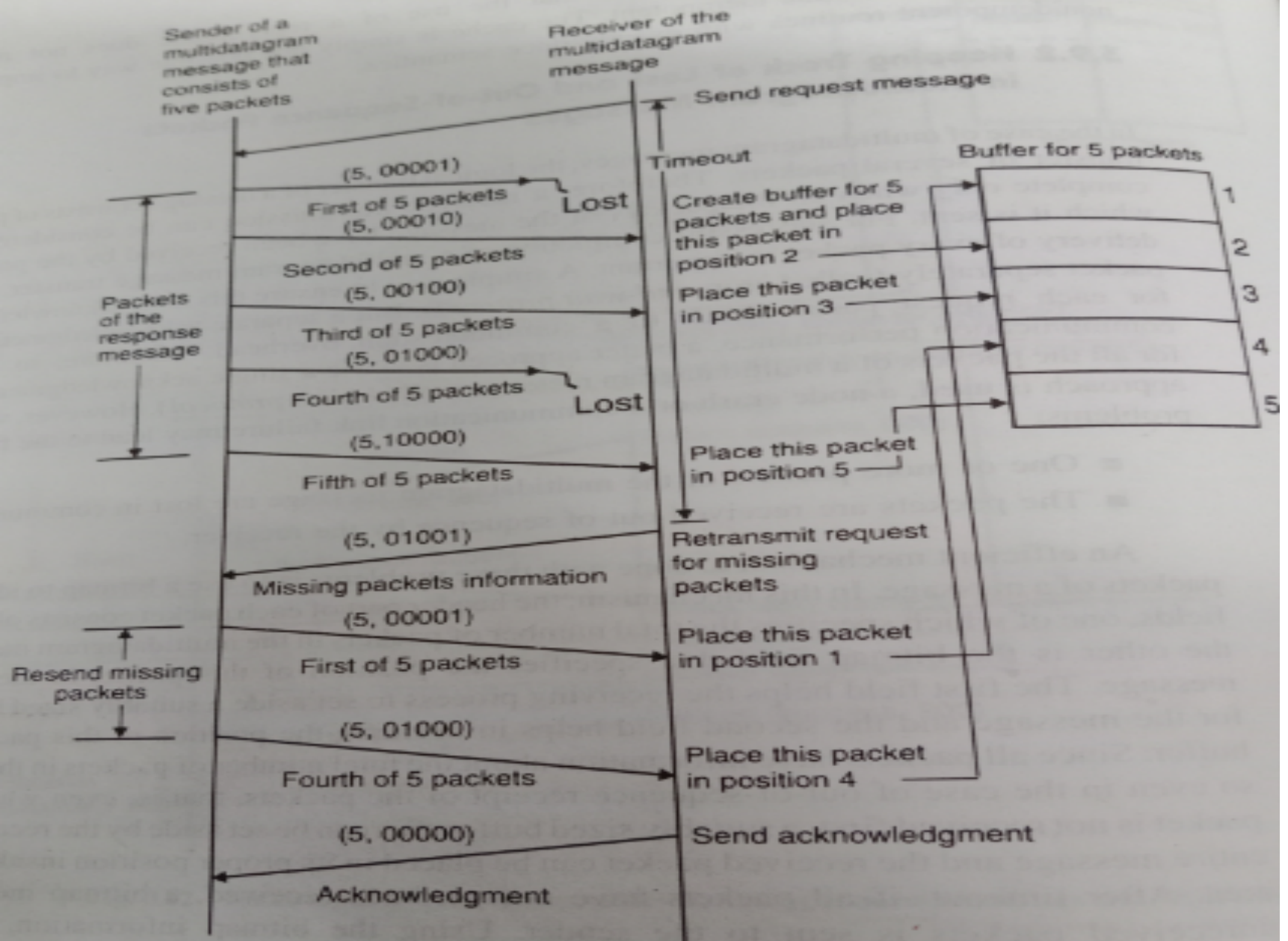


Fig. 3.13 An example of the use of a bitmap to keep track of lost and out of sequence packets in a multidatagram message transmission.