Communications in Distributed Systems

UNIT-2: Communications in Distributed Systems

- Basics of Communication Networks
- Layered Protocols
- ATM Models
- Client-Server Models
- **Blocking and Non-Blocking Primitives**
- **Buffered and Unbuffered Primitives**
- Reliable and Unreliable Primitives
- Message Passing
- Remote Procedure Calls

Blocking versus Nonblocking Primitives

- **Message Passing**: A message-passing system gives a collection of message-based IPC (Inter-Process Communication) protocols.
- The **send()** and **receive()** communication primitives are used by processes for interacting with each other.
- **For example**, Process A wants to communicate with Process B then Process A will send a message with send() primitive and Process B will receive the message with receive() primitive.
- **Synchronization Semantics**: The following are the two ways of message passing between processes:
- Blocking (Synchronous)
- Non-blocking (Asynchronous)

Cont..

- In case of blocking primitive (also called as **synchronous primitives**), when a process calls ***send*** it specifies a destination and a buffer to send to that destination.
- While the message is being sent, the sending process is **blocked** (i.e., suspended).
- The instruction following the call to *send* is not executed until the message has been completly sent, as shown in figure below.
- Similarly, a call to ***receive*** does not return control until the message has actually been received and put in the message buffer.

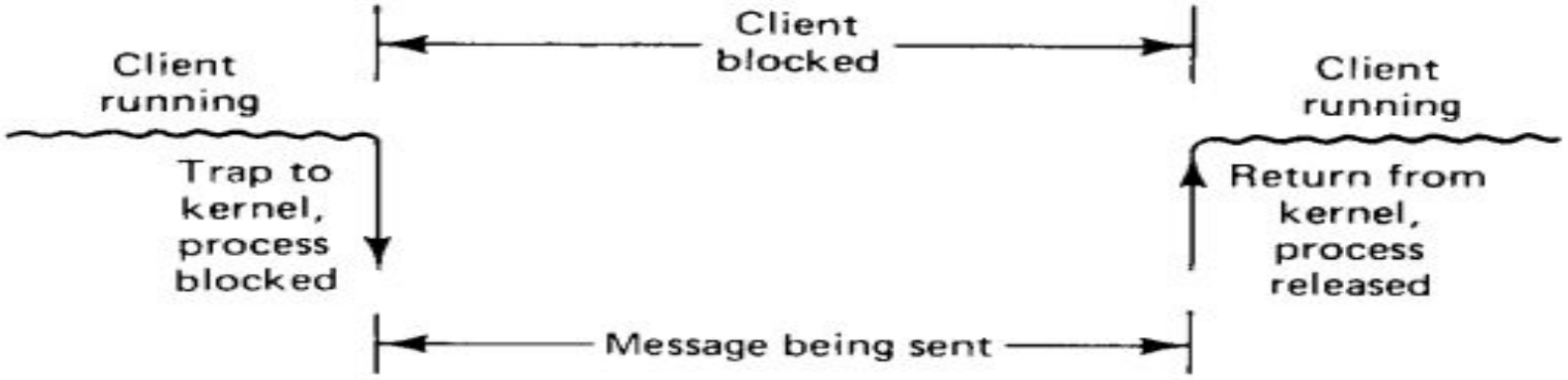

Figure 1: A blocking *send* primitive

Cont..

- An alternative to blocking primitives are **nonblocking primitives** (also called as **asynchronous primitives**).
- If ***send*** is nonblocking, it returns control to the caller immediately, before the message is sent.
- The **advantage** of this scheme is that the sending process can continue computing in parallel with the message transmission, instead of having the CPU go idle.
- The disadvantage of this scheme is that the sender cannot modify the message buffer until the message has been sent.
- The sending process has no idea of when the transmission is done, so it never knows when it is safe to reuse the buffer.
- **There are two possible solutions to this problem:**
- The **first solution** is to have the **kernel copy the message to an internal kernel buffer** and then allow the process to continue, as shown in Figure 2 in the next slide.
- The disadvantage of this method is that every outgoing message has to copied from user space to kernel space.

Cont..

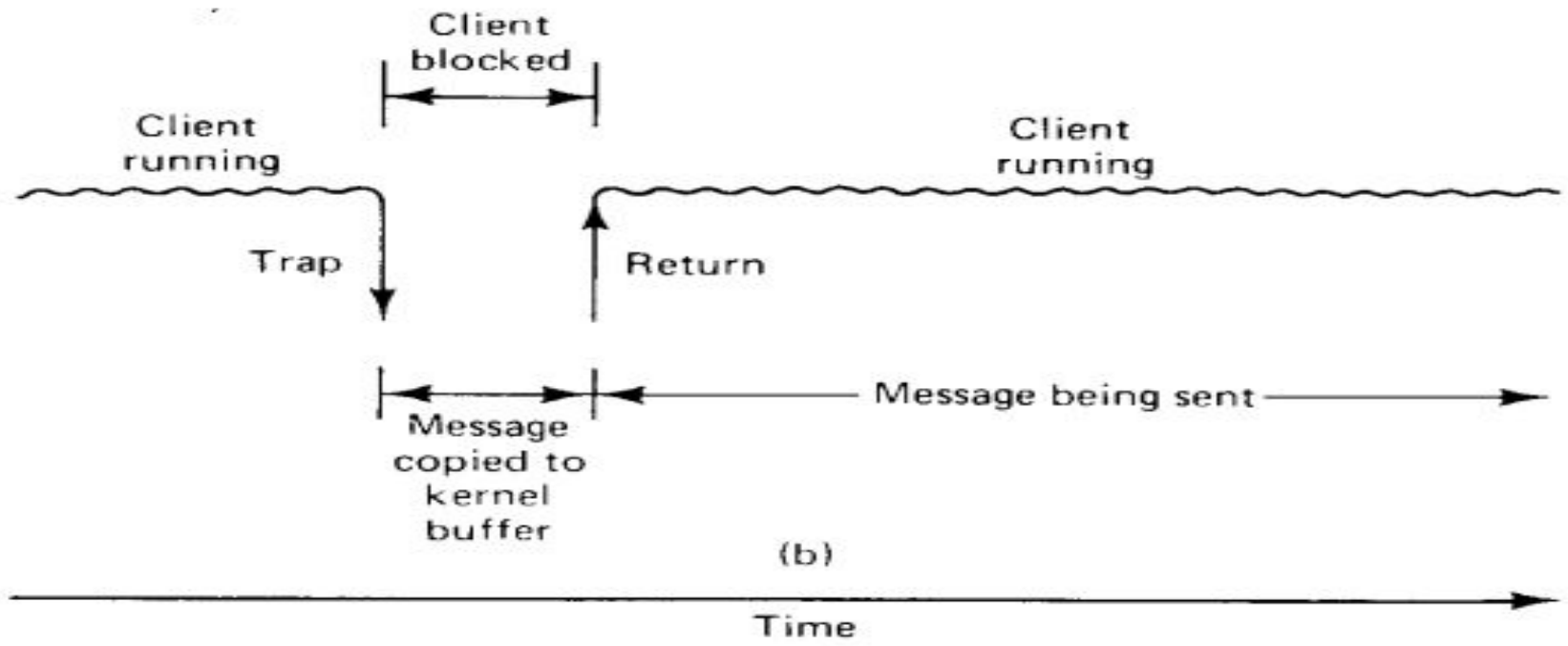

Figure 2: A nonblocking *send* primitive

Cont..

- The **second solution** is to **interrupt the sender** when the message has been sent to inform it that the buffer is once again available.
- No copy is required here, which saves time, but programs based on user-level interrupts are difficult to write and debug.

- **Blocking and nonblocking *send* primitives:**
- **Blocking send() primitive:** The blocking send() primitive refers to the blocking of sending process.
- The process remains blocking until it receives an acknowledgment from the receiver side that the message has been received after the execution of this primitive.
- **Non-blocking send() primitive:** The non-blocking send() primitive refers to the non-blocking state of the sending process that implies after the execution of send() statement, the process is permitted to continue further with its execution immediately when the message has been transferred to a buffer.

Cont..

- Just as ***send*** can be blocking and nonblocking, so can ***receive***.
- **Blocking receive() primitive:** when the receive statement is executed, the receiving process is halted until a message is received.
- **Nonblocking receive() primitive:** The non-blocking receive() primitive implies that the receiving process is not blocked after executing the receive() statement, control is returned immediately after informing the kernel of the message buffer's location.

- **The issue in a nonblocking *receive()* primitive is when a message arrives in the message buffer, how does the receiving process know?**
- One of the following two procedures can be used for this purpose:
- **Polling**: In the polling method, the receiver can check the status of the buffer when a test primitive is passed in this method.
- The receiver regularly polls the kernel to check whether the message is already in the buffer.

Cont..

- **Interrupt**: A software interrupt is used in the software interrupt method to inform the receiving process regarding the status of the message i.e. when the message has been stored into the buffer and is ready for usage by the receiver.
- So, here in this method, receiving process keeps on running without having to submit failed test requests.

Buffered versus Unbuffered Primitives

- **Unbuffered Primitives:**
- Unbuffered primitives involve direct communication without any intermediate storage.
- In these primitives, the sender and receiver need to be synchronized for the communication to take place.
- A call to the primitive ***receive(addr, &m)*** tells the kernel of the machine on which it is running that the calling process is listening to the address *addr* and is prepared to receive one message sent to that address.
- A single message buffer, pointed to by *m*, is provided to hold the incoming message.
- When the message comes in, the receiving kernel copies it to the buffer and unblocks the receiving process, as shown by Figure 3 in the next slide.

Cont..

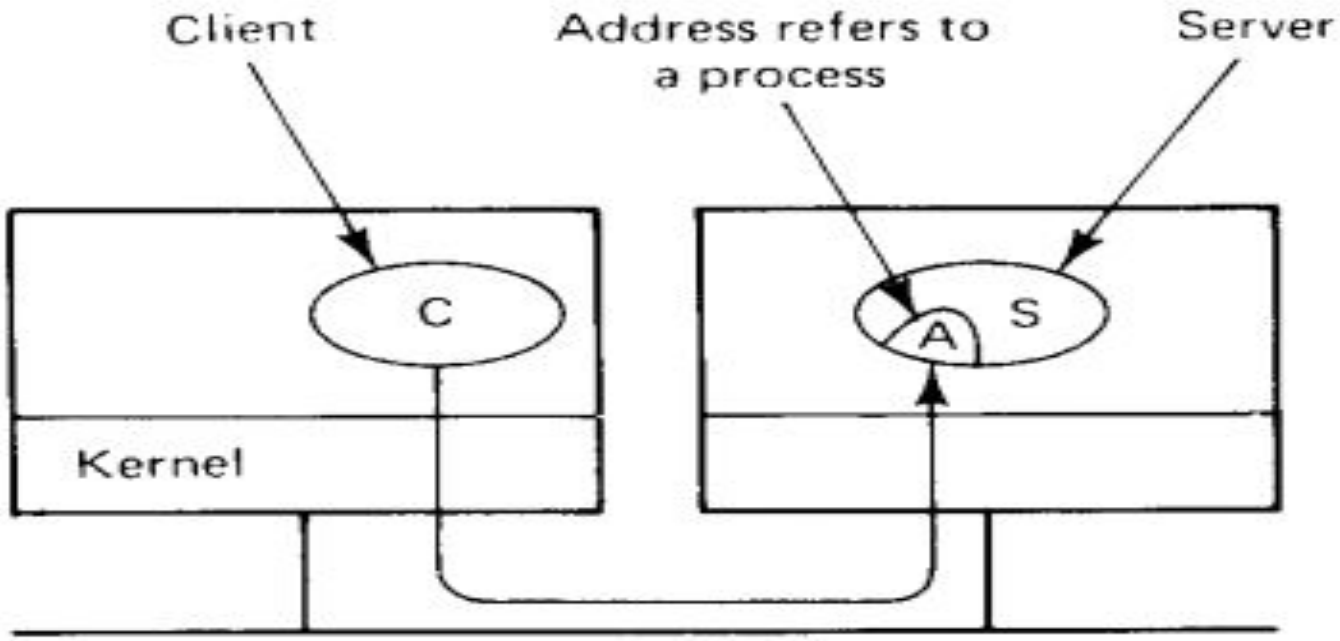

Figure 3: Unbuffered message passing

Cont..

- What are the problems that occurs when the client calls *send* primitive before the server calls *receive* primitive in unbuffered message passing mechanism?
- **The problems are as follows:**
- How does the server's kernel knows which of its process is using the address in the newly arrived message?
- How does the server's kernel knows where to copy the message?
- To avoid such problems, it's crucial to ensure that the *receive* primitive is called in a timely manner. Some strategies to handle this include:
- **Pre-emptive Design**: Design the system so that the *receive* is always invoked before or concurrently with *send* to avoid blocking.
- **Timeouts and Error Handling**: Implement timeouts or error handling mechanisms to manage situations where a *send* operation might block indefinitely.
- **Buffered Communication:** Use buffered message passing where messages are stored in a buffer temporarily, allowing the sender and receiver to operate asynchronously and reducing the risk of blocking.

Cont..

- **Buffered primitives:**
- In order to deal with the buffer management issues, a new data structure called a "**mailbox**" is defined.
- A process that is interested in receiving messages tells the kernel to create a mailbox for it, and specifies an address to look for the network packets.
- Henceforth, all incoming messages with that address are put in the mailbox.
- The call to receive removes on message from the mailbox, or blocks if none is present.
- In this way, the kernel knows what to do with incoming messages and has a place to put them.
- This technique is referred as **buffered primitive**, as shown by Figure 4, in the next slide.

Cont..

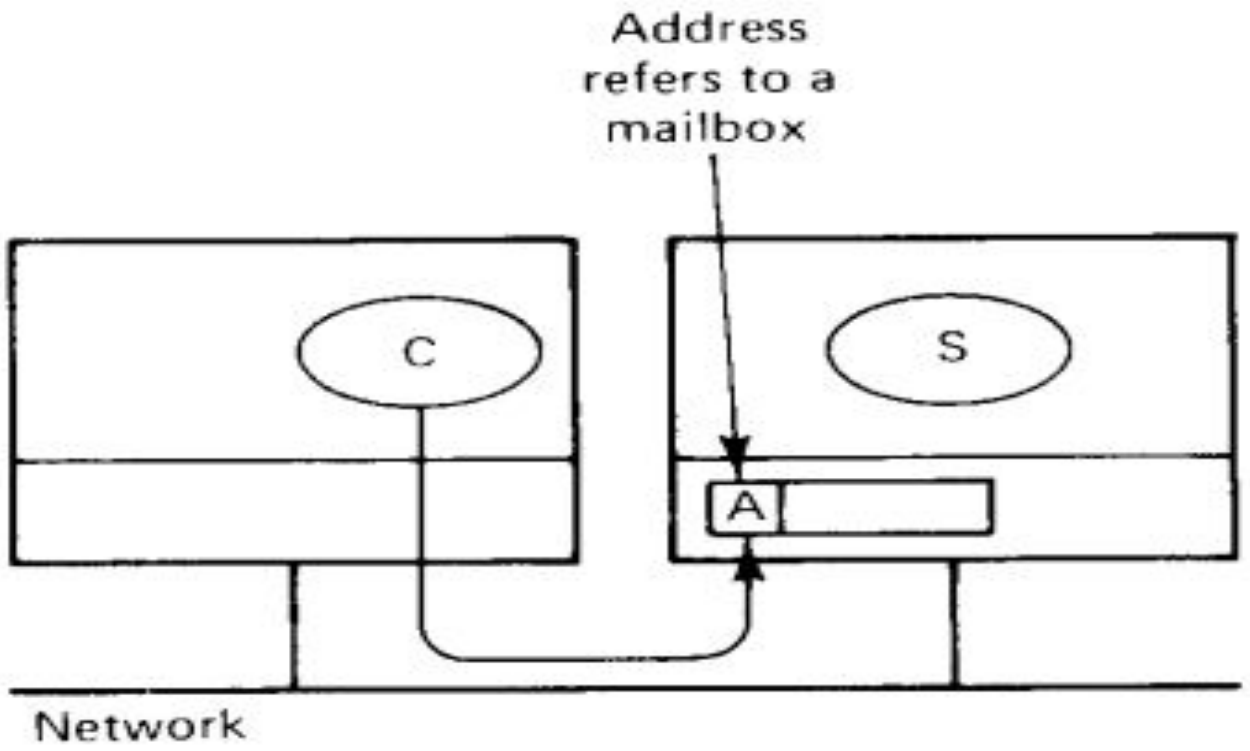

Figure 4: Buffered message passing