Remote Procedure Call - Performance

UNIT-2: Communications in Distributed Systems

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

#9: Remote Procedure Call - Performance

- **Success** or **failure** of a Distributed System **depends on** its **Performance.**
  - Performance is critically dependent on the *"Speed of the Communication*".*
- **Implementation** of Distributed System
  - Protocol Selection
  - Acknowledgement
  - Critical Path
  - Copying
  - Timer Management

#9.1:RPC Protocol Selection

- Criteria of Selection: It should gets the bits from the client's kernel to the server's kernel
- Connection-oriented protocol vs Connectionless protocol
- Standard general-purpose protocol vs Specifically designed for RPC

#9.1.1:Connection-oriented Vs Connectionless Protocol

Connection-Oriented Protocol	Connectionless Protocol
After connection establishment, the client is bound to the server	No principle of connection establishment for long period. However session-wise pairing between two neighboring entities is required.
Same connection is used by all the traffic, in both directions.	The path used by all the traffic might be different
Communication is easier	Communication is easier is LAN, where most of the connections are of one hop length
When a kernel sends a message, the possibility of lost of the message and receiving of its ACK is not worrisome for it.	Loss of message, loss of ACK need extra work
This approach is very strong in WAN	Reliable in LAN
This is not suitable in LAN	Suitable in small building LAN
Conclusion: Connection-oriented in WAN	**Conclusion**: Connectionless in LAN

#9.1.2:Standard Protocol Vs Specialized RPC

Standard Protocol (IP or UDP)	Specialized RPC
The protocol is already designed. Saves substantial work.	It is need to be invented, implemented, tested and embedded in existing systems. Considerably more work.
Many implementations are available. Saves work and time.	More work and time
Communication is easier	Communication need to be tested in the networks.
Most of the UNIX systems accept the packets of these protocol for communication purpose	Needs integration into existing UNIX systems
Existing networks also support IP and UDP packets	Need to be tested across all types of networks
Writing, executing and testing code using these protocols are straightforward.	Several phases of software testing is required.
IP is not an end-to-end protocol. It is executed on top of reliable TCP. So, it bounce back several times in the network.	Specialized RPC would avoid bouncing back of the packets.
IP has 13 header fields. 3 are essential (Src_Addr, Dstn_addr, Pkt_len). Header checksum is time consuming	Number of header fields may differ, according to the requirement of the problem.

#9.2:Acknowledgements

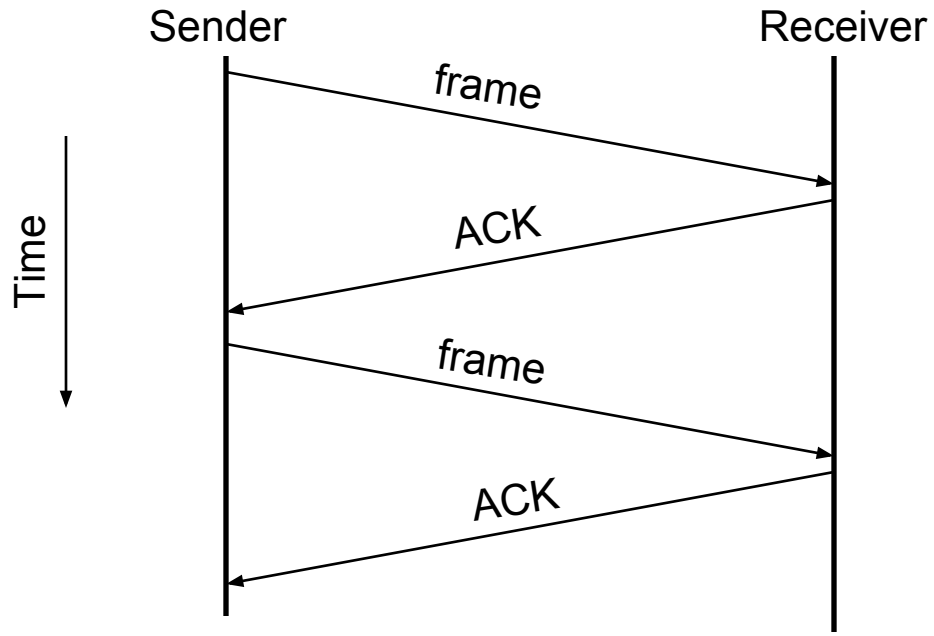
- When large RPCs have to be broken up into many small packets, then what should be the acknowledgement process?
  - Should individual packets be acknowledged? (***stop-and-wait- protocol***)
  - Acknowledge after receiving all the packets (***Blast Protocol***)
- *******
- *******

Automatic Repeat Request (ARQ) Algorithms

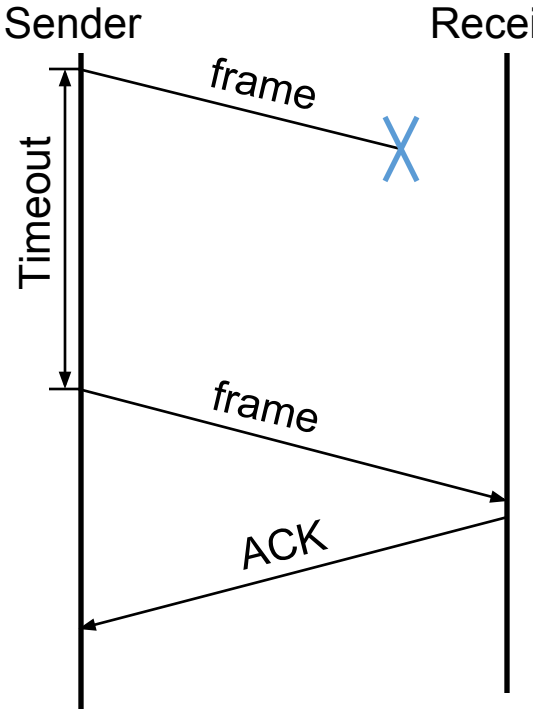
- Use two basic techniques:
  - Acknowledgements (ACKs)
  - Timeouts
- Two examples:
  - Stop-and-Wait
  - Sliding window

Stop-and-Wait

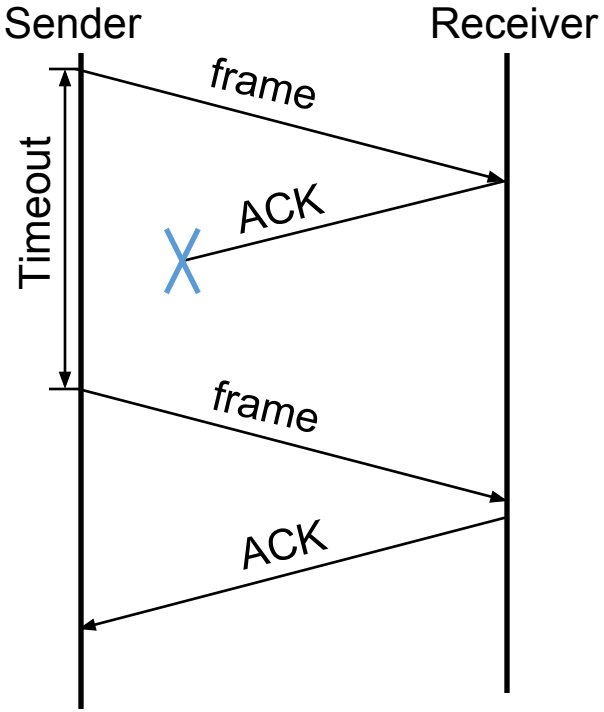
- Receiver: send an acknowledge (ACK) back to the sender upon receiving a packet (frame)
- Sender: excepting first packet, send a packet only upon receiving the ACK for the previous packet



What Can Go Wrong?

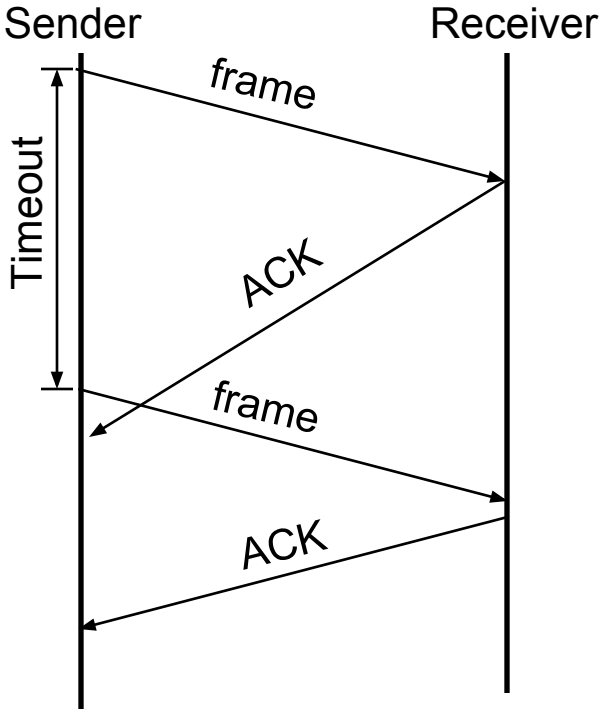


Frame lost □ resent it on Timeout



ACK lost □ resent packet

Need a mechanisms to detect duplicate packet

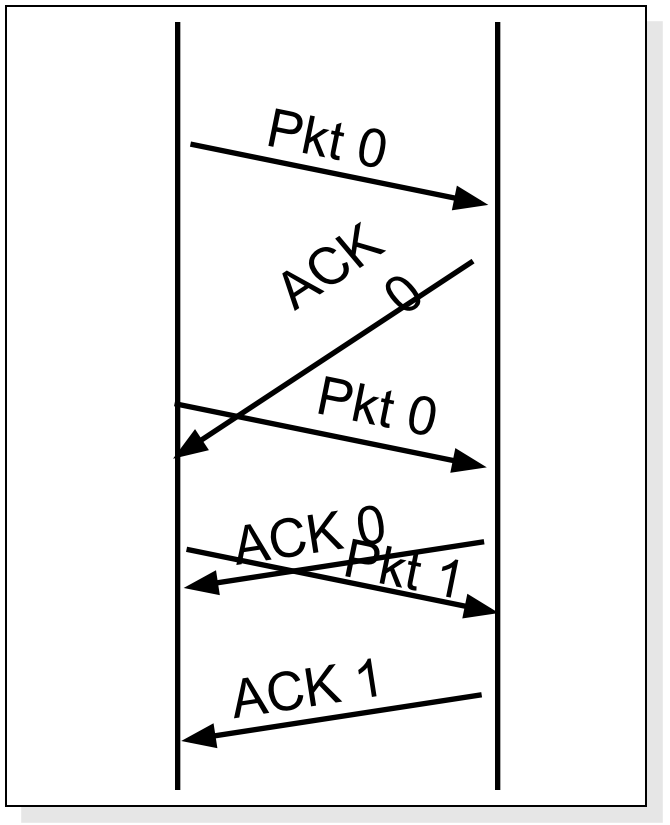


ACK delayed □ resent packet

Need a mechanism to differentiate between ACK for current and previous packet

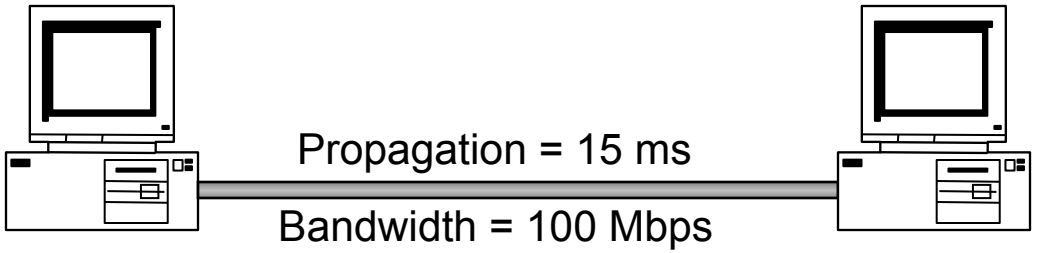
How to Recognize Retransmissions?

- Use sequence numbers
  - both packets and acks
- Sequence # in packet is finite ? How big should it be?
  - For stop and wait?
- One bit – won't send seq #1 until received ACK for seq #0



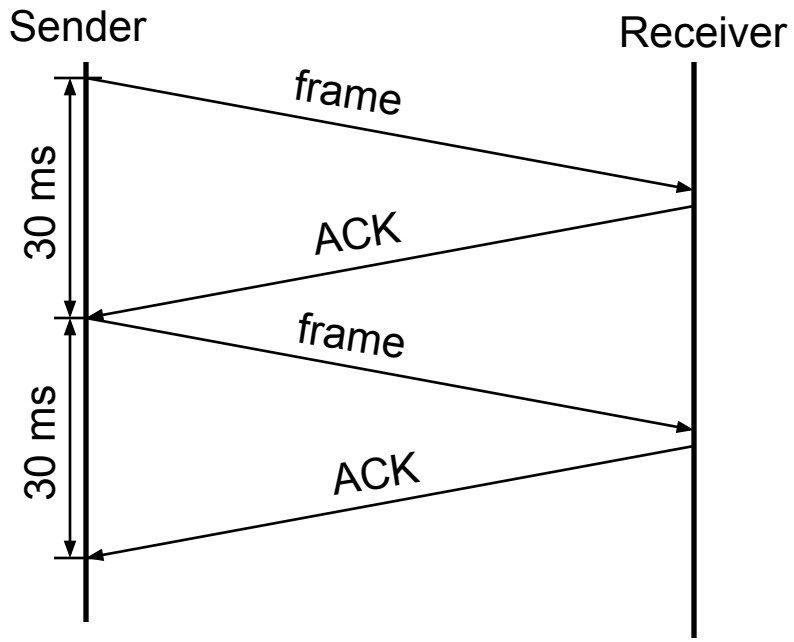
Stop-and-Wait Disadvantage

- May lead to inefficient link utilization
- Examplems: assume
  - One-way propagation = 15 ms
  - Bandwidth = 100 Mbps
  - Packet size = 1000 bytes transmit = $(8*1000)/10^8$ = 0.08ms
  - Neglect queue delay Latency = approx. 15 ms; RTT = 30



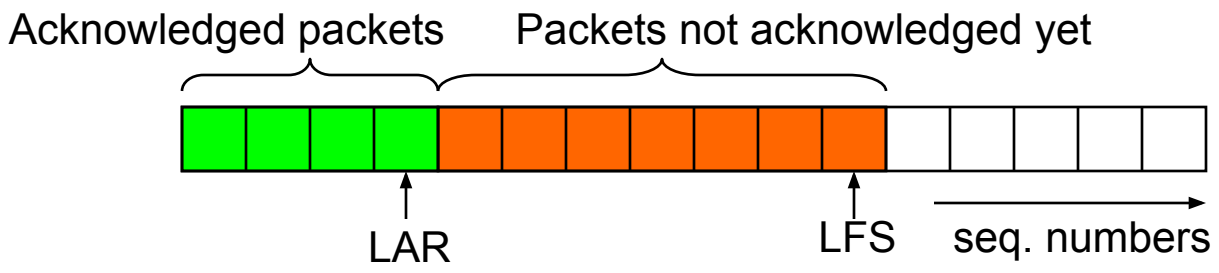
Stop-and-Go Disadvantage cont.

- Send a message every 30 ms Throughput = (8*1000)/0.03 = 0.2666 Mbps
- Thus, the protocol uses less than 0.3% of the link capacity!



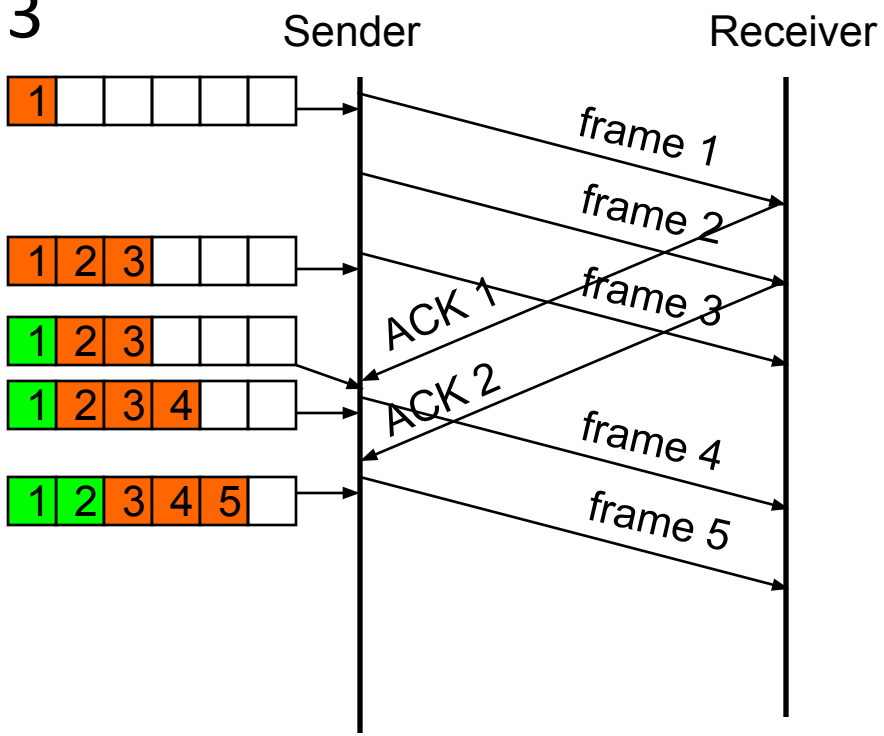
Sliding Window Protocol: Sender

- Each packet has a sequence number
  - Assume infinite sequence numbers for simplicity
- Sender maintains a window of sequence numbers
  - **SWS** (sender window size) – maximum number of packets that can be sent without receiving an ACK
  - **LAR** (last ACK received)
  - **LFS** (last frame sent)



Example

- Assume SWS = 3



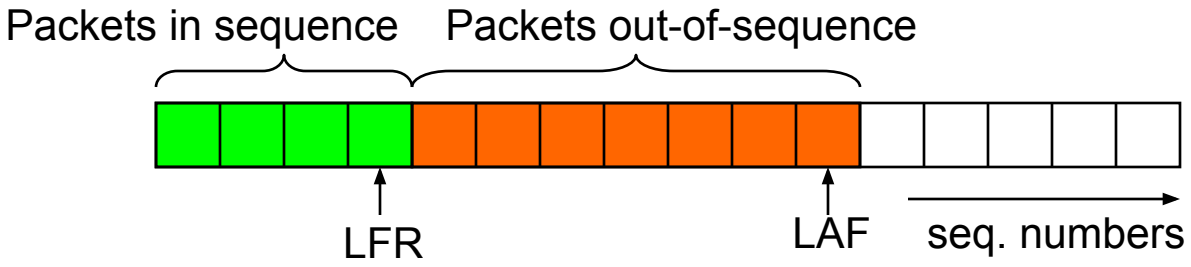
Note: usually ACK contains the sequence number of the first packet in sequence expected by receiver

Sliding Window Protocol: Receiver

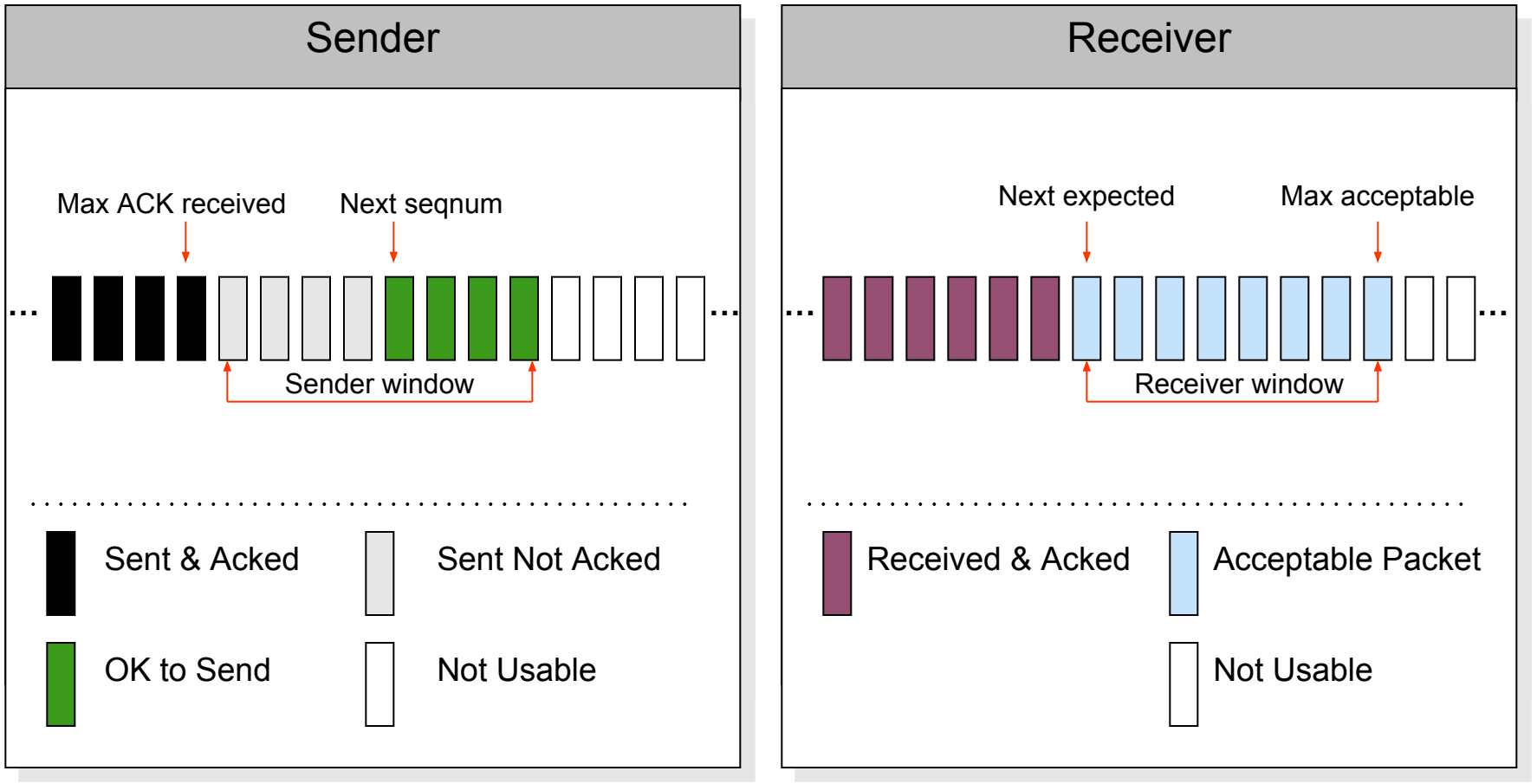
- Receiver maintains a window of sequence numbers
  - RWS (receiver window size) – maximum number of out-of-sequence packets that can received
  - LFR (last frame received) – last frame received in sequence
  - LAF (last acceptable frame)
  - LAF – LFR <= RWS

Sliding Window Protocol: Receiver

- Let seqNum be the sequence number of arriving packet
- If (seqNum <= LFR) or (seqNum >= LAF)
  - Discard packet
- Else
  - Accept packet
  - ACK largest sequence number seqNumToAck, such that all packets with sequence numbers <= seqNumToAck were received



Sender/Receiver State



Sequence Numbers

- How large do sequence numbers need to be?
  - Must be able to detect wrap-around
  - Depends on sender/receiver window size
- E.g.
  - Max seq = 7, send win=recv win=7
  - If pkts 0..6 are sent succesfully and all acks lost
    1. Receiver expects 7,0..5, sender retransmits old 0..6!!!
- Max sequence must be ≥ send window + recv window

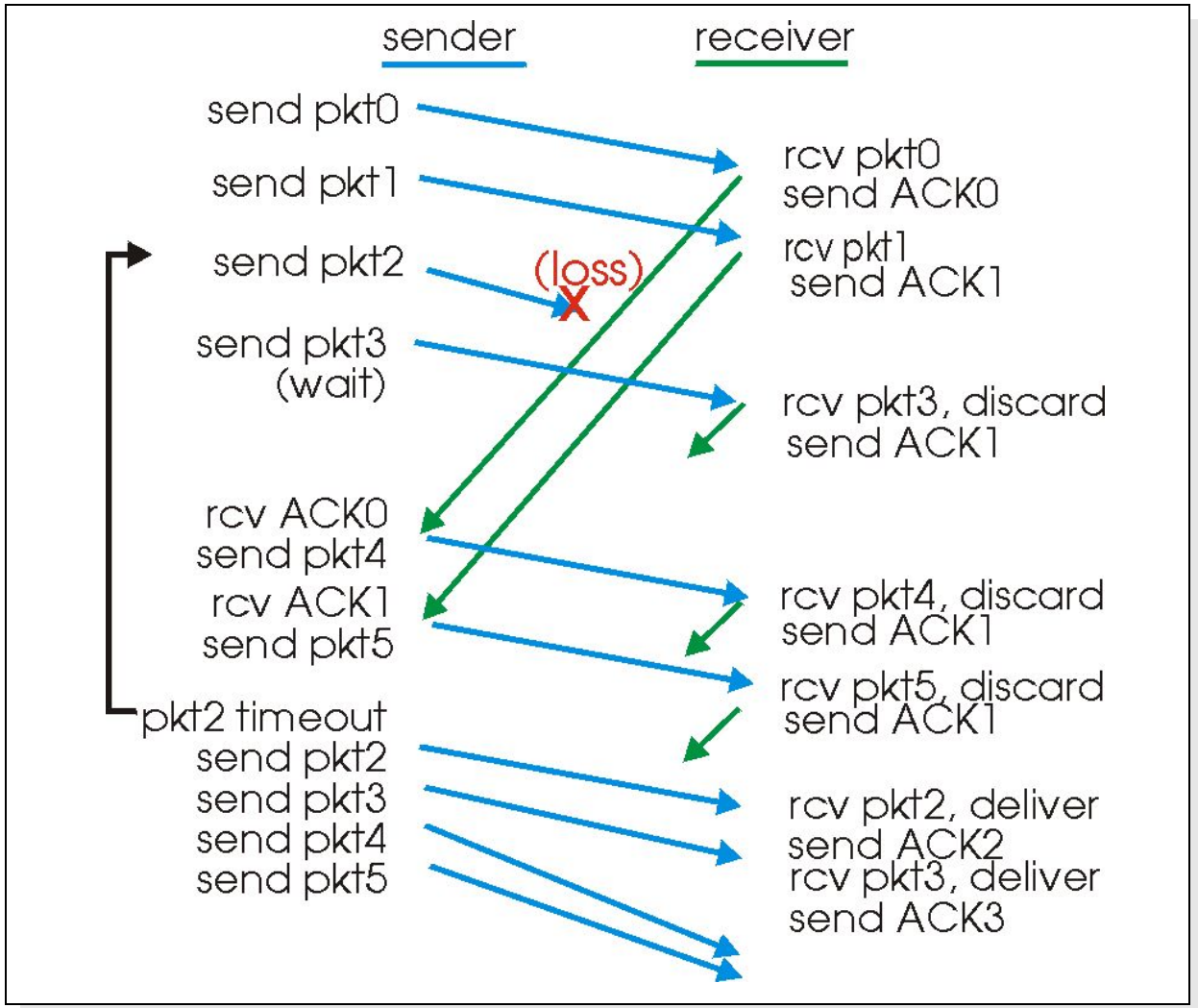
Cumulative ACK + Go-Back-N

- On reception of new ACK (i.e. ACK for something that was not acked earlier)
  - Increase sequence of max ACK received
  - Send next packet
- On reception of new in-order data packet (next expected)
  - Hand packet to application
  - Send cumulative ACK – acknowledges reception of all packets up to sequence number

Loss Recovery

- On reception of out-of-order packet
  - Send nothing (wait for source to timeout)
  - Cumulative ACK (helps source identify loss)
- Timeout (Go-Back-N recovery)
  - Set timer upon transmission of packet
  - Retransmit all unacknowledged packets

Go-Back-N in Action

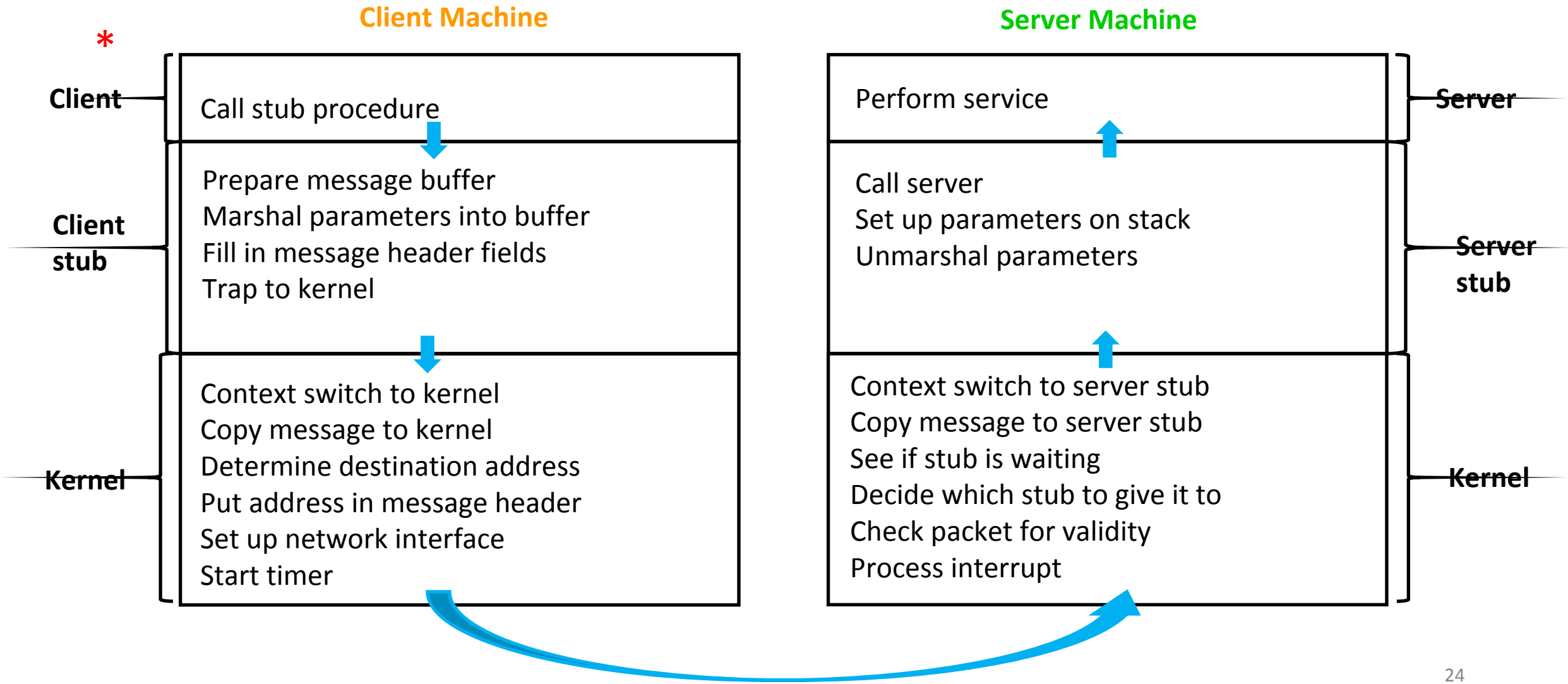


#9.3: Critical Path

- A critical path is defined as the sequence of instructions that is executed on every RPC (Eg. A client to a remote server)
- There are 14 steps in the RPC from Client-to-Server

1. Call stub
2. Get message buffer
3. Marshal parameters
4. Fill in headers
5. Compute UDP checksum
6. Trap to Kernel
7. Queue packet for transmission
8. Move packet to controller over the QBus
9. Ethernet transmission time
10. Get packet from controller
11. Interrupt service routine
12. Computer UDP Checksum
13. Context switch to user space
14. Server stub code

#9.3: Critical Path : Schematic View

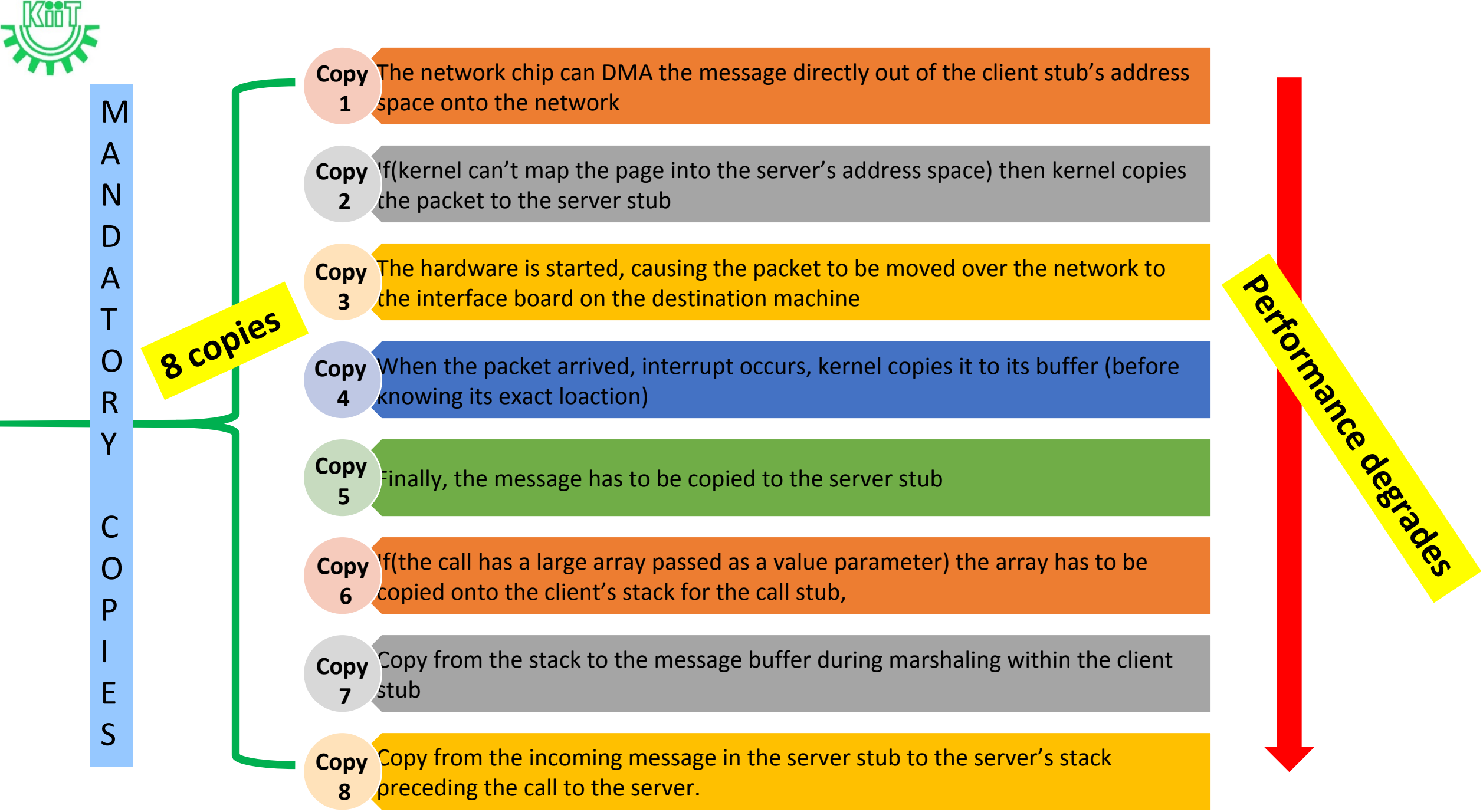


#9.3: Critical Path Cont.

- **Q:** Where is most of the time spent on the critical path?
- **Ans:**
  - Marshaling parameters and moving messages around
  - In case of null RPC, context switch to the server stub when packet arrives, the interrupt service routine, and moving the packets to the network interface for transmission
  - Managing a pool of buffers – which client stubs use to avoid having to fill in the entire UDP header every time.
  - All the machines don't share the same address space, so context switch and use of page table takes time
  - Entire RPC system has been carefully coded in assembly language and hand optimized. So, it is faster and saves time.
- *****

#9.4: Copying

- Copy is an issue in RPC [∵ it dominates execution time in RPC]
- **Q:** Why does this issue occur in RPC? [∵ In most of the systems the kernel and the user address spaces are disjoint. A message must be copied 1-to-8 times depending on the h/w, s/w and type of call].
- **Analysis:** In general a message is required to be copied many times during RPC communication. It hampers the performance of the execution time of the RPC.
- The 8-different copies degrades the performance of RPC.


MANDATORY COPIES
8 copies
Copy 1
The network chip can DMA the message directly out of the client stub's address space onto the network
Copy 2
If(kernel can't map the page into the server's address space) then kernel copies the packet to the server stub
Copy 3
The hardware is started, causing the packet to be moved over the network to the interface board on the destination machine
Copy 4
When the packet arrived, interrupt occurs, kernel copies it to its buffer (before knowing its exact loaction)
Copy 5
Finally, the message has to be copied to the server stub
Copy 6
If(the call has a large array passed as a value parameter) the array has to be copied onto the client's stack for the call stub,
Copy 7
Copy from the stack to the message buffer during marshaling within the client stub
Copy 8
Copy from the incoming message in the server stub to the server's stack preceding the call to the server.
Performance degrades

#9.4: Copying Cont.

- How to eliminate unnecessary copying?
  - Using the hardware ***scatter-gather***
  - **At the Sender's side**: With cooperative hardware, a reusable packet header inside the kernel and a data buffer in user space can be put out onto the network with no internal copying on the sending side.
  - **At the Receiver's side**: Dump the message into a kernel buffer and let the kernel figure out what to do with it.
  - **In Operating Systems:** Using virtual memory
  - **Using mapping:** If(memory map can be updated in less time)
    1. Then, mapping is faster than copying
    2. Else, Not
- *****

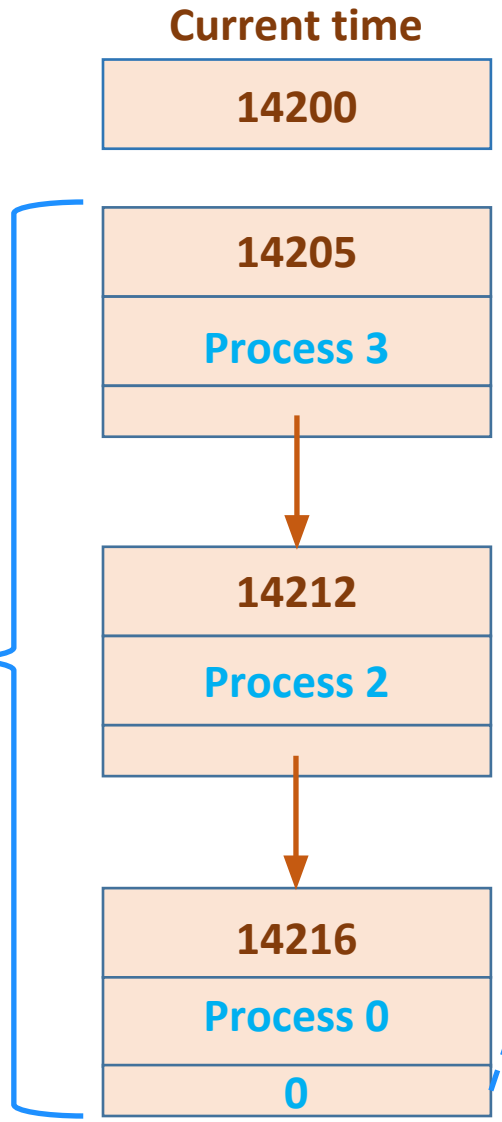
#9.5: Timer Management

- **Timer**: It is an automatic mechanism for activating an entity at a preset time.
- **Setting a timer** requires **building a data structure** specifying when the timer is to expire and what is to be done when that happens.
- The list of messages are in sorted order
- Timer starts just after message transmitted
- If(ACK or Reply arrives before the timer expires)
  - Then the timeout entry must be located and removed from the list
- Timer value should be neither too high or too low
- Most systems maintains a **Process Table** to implement **Timer**

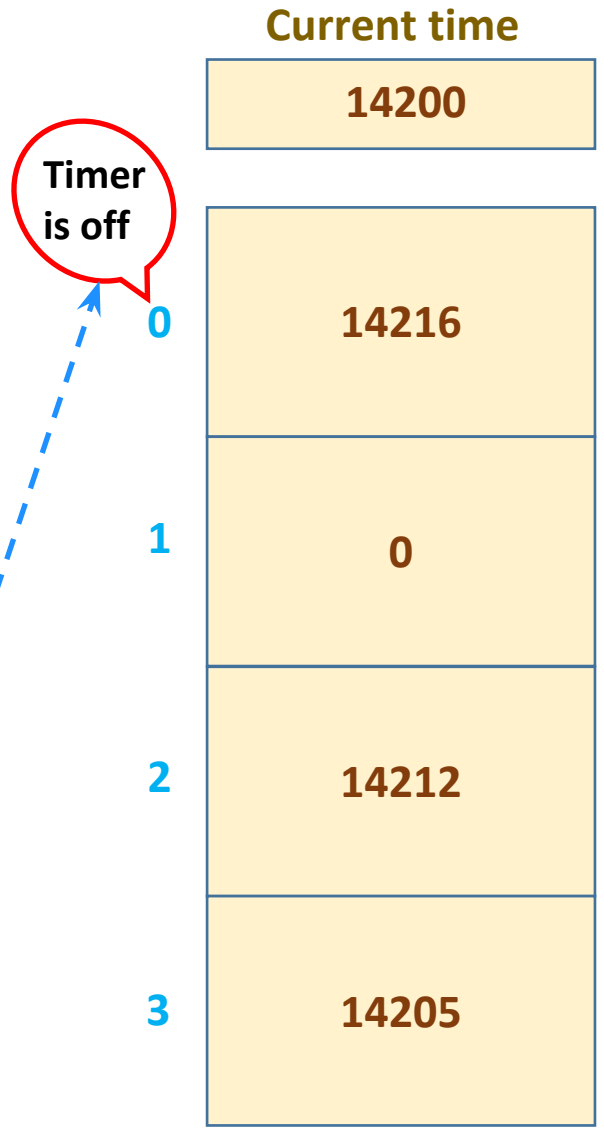
#9.5: Timer Management via Sorted List and Process Table

- *

Sorted List w.r.t. time out



(a) Timeouts in a sorted list



(b) Timeouts in a process table

Explanation: In process table in stead of storing timeouts in a sorted linked list, each process table entry has a field for holding its timeouts. It is shown the left of the process table in **blue color.**

Working Principle: The kernel scans the entire process table, checks each timer value against the current time. If (T_{read}<=$T_{current}$) then it is processed and reset.

Note: *Sweep Algorithms* operates by periodically making a sequential pass through a process table.

Q&A

Practice Questions

1) How an Operating System avoid copy of a message during RPC? (Pg-93)
2) Suppose that the time to do a null RPC (i.e., 0 data bytes) is 1.0 msec, with an additional 1.5 msec for every 1K of data. How long does it take to read 32K from the file server in a single 32K RPC? How about as 32 1K RPCs? (Pg. 117)
3)

Thank You!