18CSE356T
Distributed Operating Systems

Unit II

CLR-2 : TO COMPREHEND ABOUT THE COMMUNICATION THAT TAKES PLACE IN DISTRIBUTED SYSTEMS

CLO-2 : CATEGORIZE LAYERED PROTOCOLS AND COMPREHEND THE COMMUNICATIONS IN DISTRIBUTED SYSTEMS

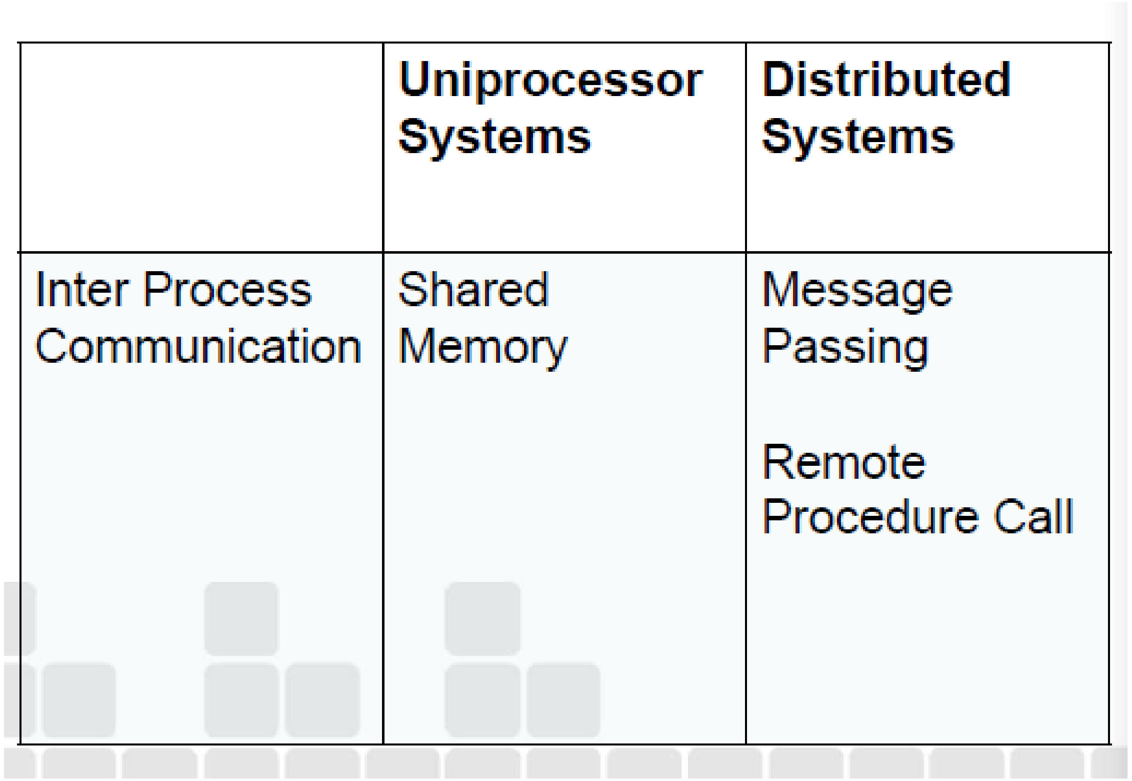
Discusses about issues, examples and problems associated with inter process communication in distributed operating systems

TOPICS COVERED

- Fundamentals of Communication systems
- Layered Protocols
- ATM networks
- Client Server model
  - Blocking Primitives
  - Non-Blocking Primitives
  - Buffered Primitives
  - Unbuffered Primitives
  - Reliable primitives
  - Unreliable primitives
- Message passing and its related issues
- Remote Procedure Call and its related issues
- Case Studies: SUN RPC, DEC RPC D

Fundamentals of Communication systems

Most important difference between a DS and a uniprocessor system is INTERPROCESS COMUNICATION.



	Uniprocessor Systems	**Distributed Systems**
Inter Process Communication	Shared Memory	Message Passing Remote Procedure Call

Layered Protocols

- In a distributed system, processes run on different machines.
- Processes can only exchange information through ***message passing.***
  - harder to program than shared memory communication
- Successful distributed systems depend on communication models that hide or simplify message passing
- Process A ←---→ Process B
- Agreements needed.
  - How receiver knows which is last bit of the message?

Layered Protocols (1)

- In 1983, Day and Zimmerman
- ISO OSI Model
- International Standards Organization (ISO) - developed a reference model called OSI (Open Systems Interconnection Model)
- OPEN SYSTEM – Is the one that is prepared to communicate with any other open system by using standard rules that govern the format, contents and meaning of the messages sent and received.

OSI Model

- The **OSI model** provides the standard for communication so that different manufacturers' computers can be used on the same network.
- The **OSI** reference **model** describes how data is sent and received over a network. This **model** breaks down data transmission over a series of seven **layers**.
- Purpose of OSI:
  - The original objective of the OSI model was to provide a set of design standards for equipment manufacturers so they could communicate with each other. The OSI model defines a hierarchical architecture that logically partitions the functions required to support system-to-

Open Systems Interconnection Reference Model (OSI)

- Identifies/describes the issues involved in low-level message exchanges
- Divides issues into 7 levels, or layers, from most concrete to most abstract
- Each layer provides an interface (set of operations) to the layer immediately above
- Supports communication between open systems
- Defines functionality – not specific protocols

Connectionless vs Connection Oriented

- Connection Oriented Protocol Sevices
  - It is the communication service in which virtual connection is created before sending the packet over the internet.
- Connectionless Protocol Sevices
  - In this communication service, packets are sent without creating any virtual connection over the internet.

Layered Protocols (2)

High level 7→

Create message, 6 → string of bits

Establish Comm. 5→

Create packets 4→

Network routing 3→

Add header/footer tag checksum 2→

Transmit bits via 1→ comm. medium (e.g. Copper, Fiber, wireless)

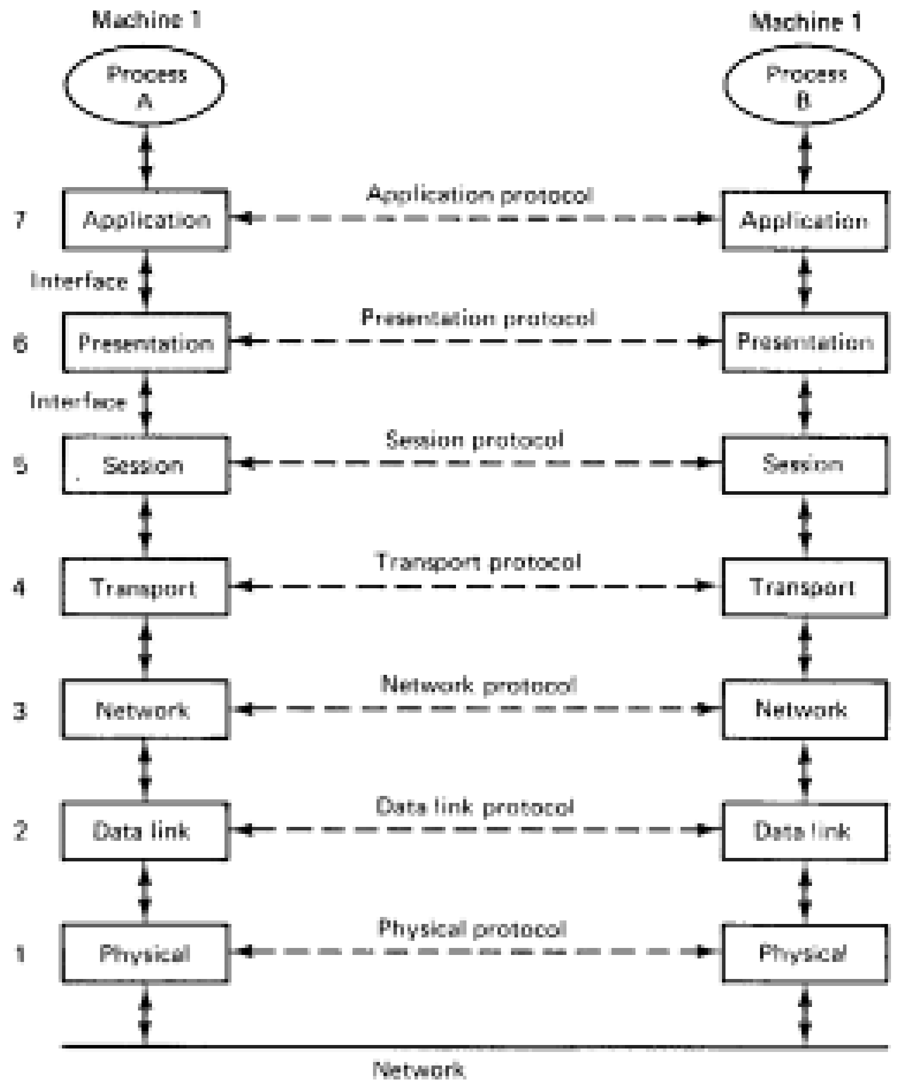


Fig. 2-1. Layers, interfaces, and protocols in the OSI model.

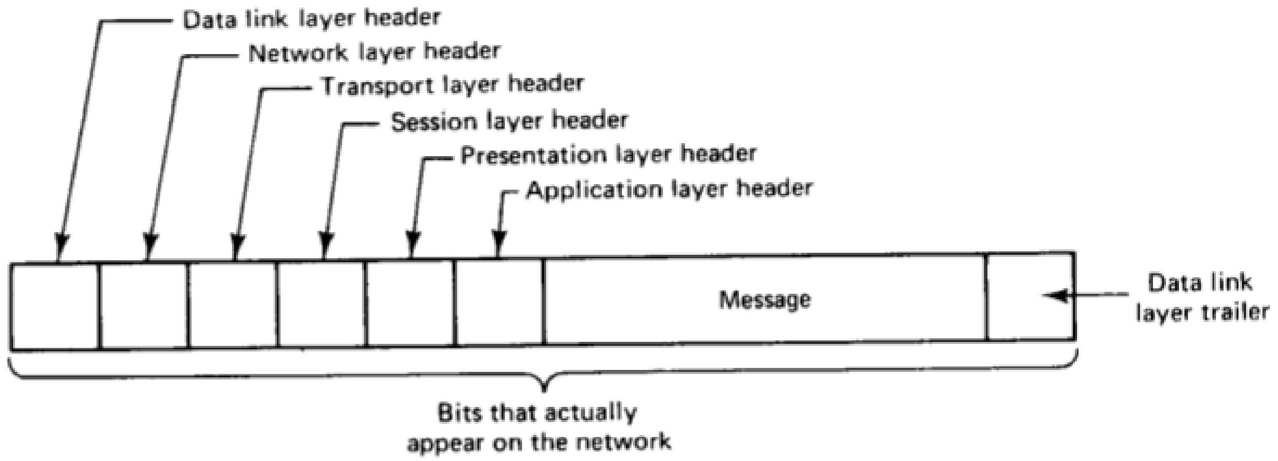
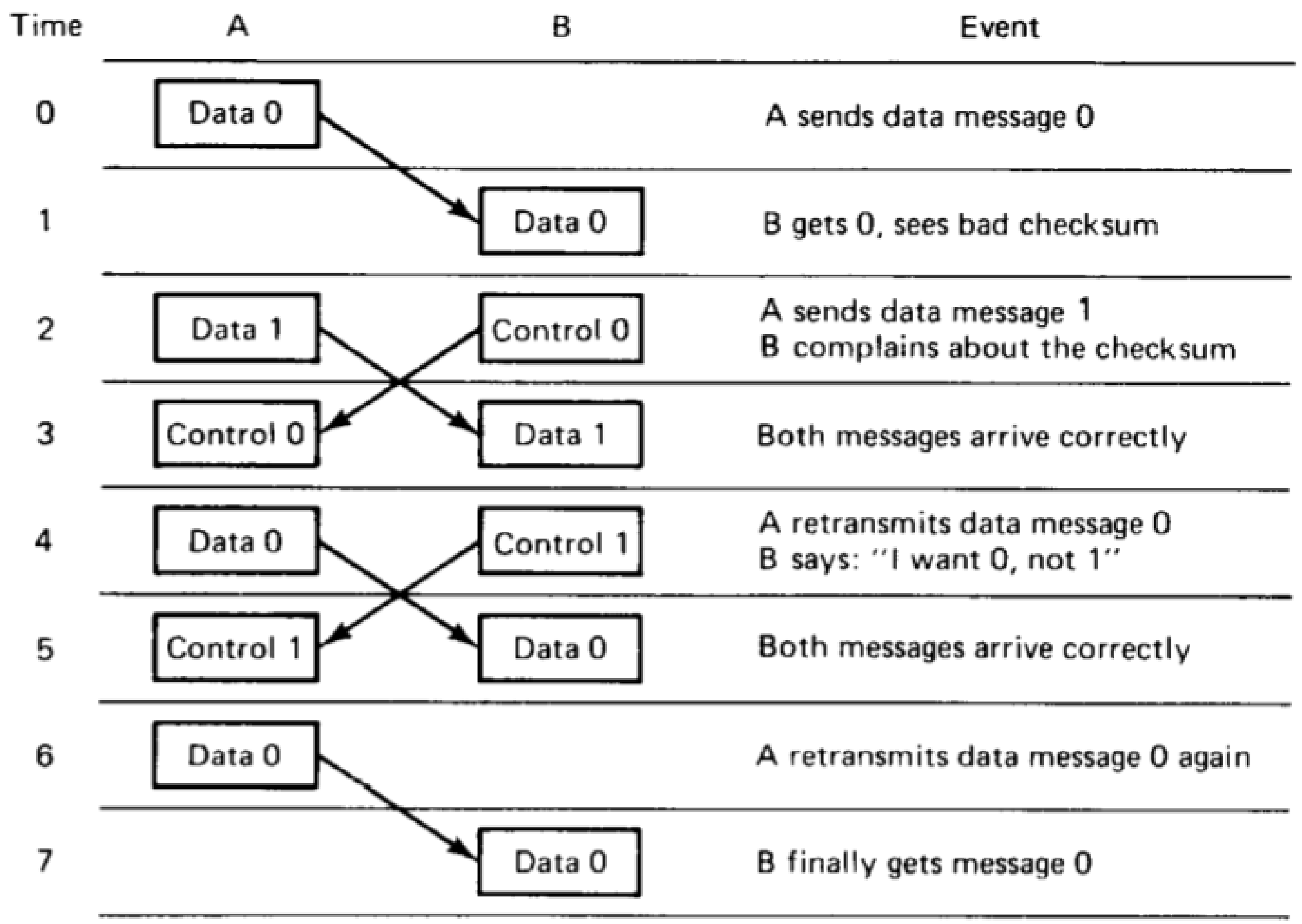


Fig. 2-2. A typical message as it appears on the network.



Time	A	B	Event
0	Data 0		A sends data message 0
1		Data 0	B gets 0, sees bad checksum
2	Data 1	Control 0	A sends data message 1 B complains about the checksum
3	Control 0	Data 1	Both messages arrive correctly
4	Data 0	Control 1	A retransmits data message 0 B says: "I want 0, not 1"
5	Control 1	Data 0	Both messages arrive correctly
6	Data 0		A retransmits data message 0 again
7		Data 0	B finally gets message 0

Fig. 2-3. Discussion between a receiver and a sender in the data link layer.

OSI Layers

7. The Application Layer

High-level application protocols, e.g., e-mail, video conferencing, file transfer, etc.

6. The Presentation Layer

- Concerned with the meaning of bits in the message
- Notifies receiver that message contains a particular record in a certain format

5. The Session Layer

- Provides dialog control and synchronization facilities
- Check points can be used so that after recovering from a crash transmission can resume from the point just before the crash
- Rarely used

OSI Layers (1)

4. The Transport Layer

•Provides a mechanism to assure the Session Layer that messages sent are all received without any data corruption or loss

•Breaks message from Session Layer into appropriate chunks (e.g., IP Packets), numbers them and sends them all

•Communicates with receiver to ensure that all have been received, how many more the receiver can receive, etc.

3. The Network Layer

Determines route (next hop) message will take to bring it closer to its destination

2. The Data Link Layer

•Detects and corrects data transmission errors (data corruption, missing data, etc.)

•Gathers bits into frames and ensures that each frame is received correctly

•Sender puts special bit pattern at the start and end of each frame + a checksum + a frame number

1. The Physical Layer

Concerned with Transmitting Bits

Lower-level Protocols

- **Physical**: standardizes electrical, mechanical, and signaling interfaces; e.g.,
  - # of volts that signal 0 and 1 bits
  - # of bits/sec transmitted
  - Plug size and shape, # of pins, etc.
- **Data Link**: provides low-level error checking
  - Appends start/stop bits to a frame
  - Computes and checks checksums
- **Network**: routing (generally based on IP)
  - IP packets need no setup
  - Each packet in a message is routed independently of the others

Transport Protocols

- **Transport layer, sender side**: Receives message from higher layers, divides into packets, assigns sequence #
- Reliable transport (connection-oriented) can be built on top of connection-oriented or connectionless networks
  - When a connectionless network is used the transport layer re-assembles messages in order at the receiving end.
- Most common transport protocols: TCP/IP

Reliable/Unreliable Communication

- TCP guarantees reliable transmission even if packets are lost or delayed.
- Packets must be acknowledged by the receiver– if ACK not received in a certain time period, resend.
- Reliable communication is considered *connection-oriented* because it “looks like” communication in circuit switched networks. One way to implement virtual circuits
- Other virtual circuit implementations at layers 2 & 3: ATM, X.25, Frame Relay, ..

Reliable/Unreliable Communication

- For applications that value speed over absolute correctness, TCP/IP provides a *connectionless* protocol: UDP
  - UDP = Universal Datagram Protocol
- Client-server applications may use TCP for reliability, but the overhead is greater
- Alternative: let applications provide reliability (end-to-end argument).

Higher Level Protocols

- **Session layer**: rarely supported
  - Provides dialog control;
  - Keeps track of who is transmitting
- **Presentation**: also not generally used
  - Cares about the meaning of the data
    - Record format, encoding schemes, mediates between different internal representations
- **Application**: Originally meant to be a set of basic services; now holds applications and protocols that don’t fit elsewhere

Middleware Protocols

- Tanenbaum proposes a model that distinguishes between application programs, application-specific protocols, and general-purpose protocols .
- Compared to the OSI model, ***the session and presentation layer have been replaced by a single middleware layer*** that contains application-independent protocols.
- These protocols do not belong in the lower layers.
- Claim: there are general purpose protocols which are ***not application specific and not transport protocols***; many can be classified as *middleware* protocols.

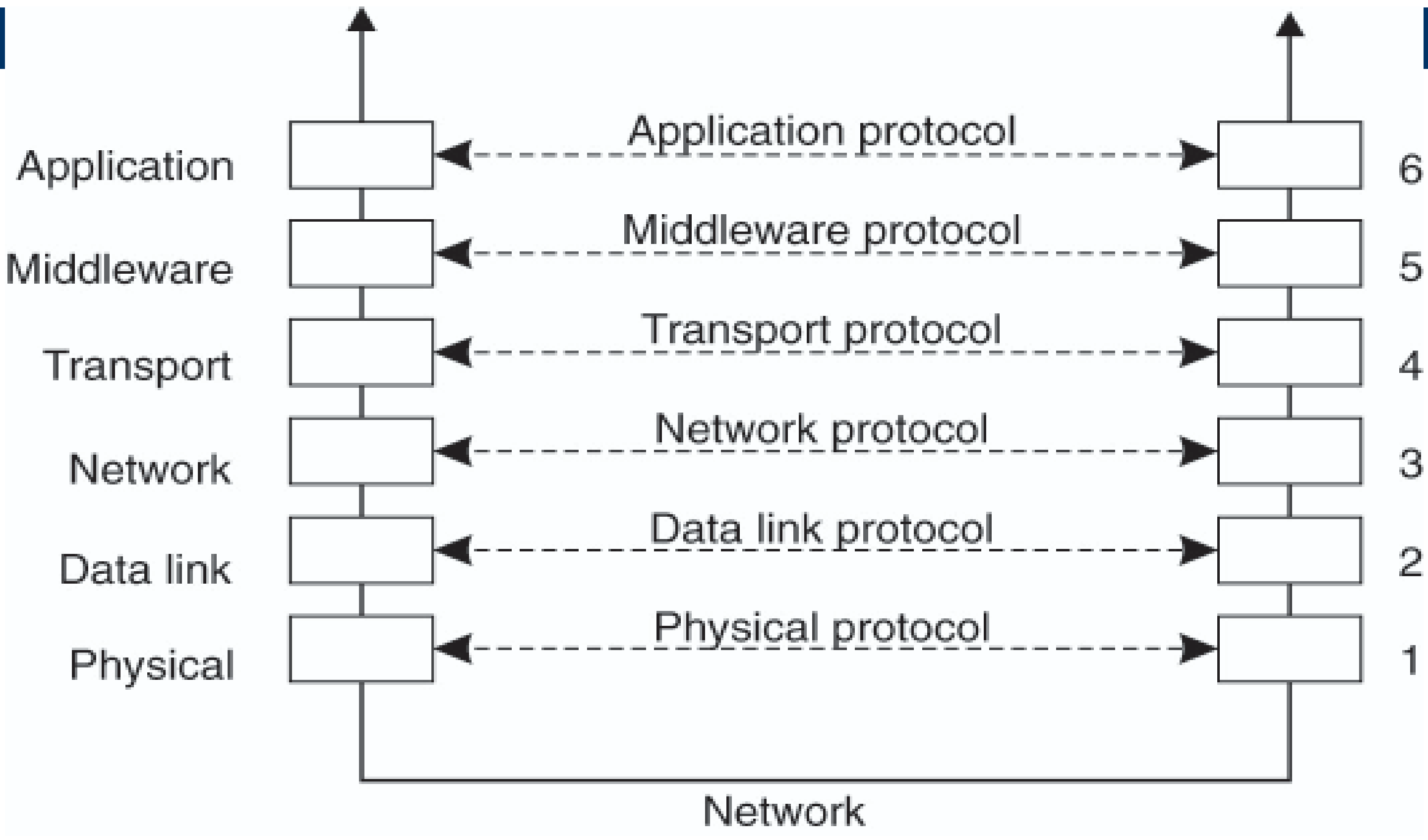


Figure 4-3. An adapted reference model for networked communication.

What is Asynchronous Transfer Mode (ATM)?

- Asynchronous Transfer Mode (ATM) is a connection-oriented, high-speed, low-delay switching and transmission technology that uses short and fixed-size packets, called cells, to transport information.
- Using the cell switching technique, **ATM combines the benefits of both circuit switching** (low and constant delay, guaranteed capacity) **and packet switching** (flexibility, efficiency for busty traffic) to support the transmission of multimedia traffic such as voice, video, image, and data over the same network.

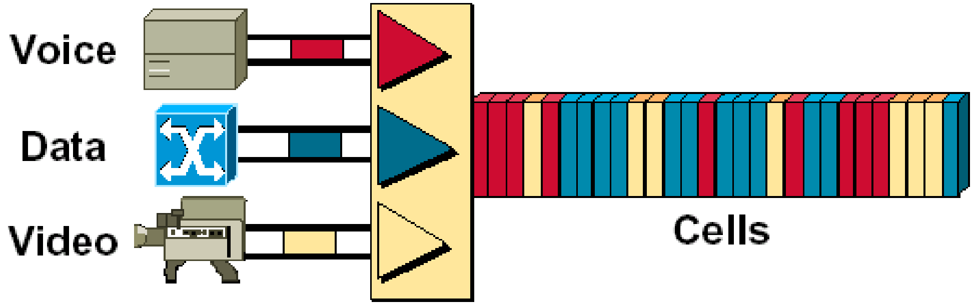
Circuit Switching v Packet Switching

- Circuit switching is *connection-oriented* (think traditional telephone system)
  - Establish a dedicated path between hosts
  - Data can flow continuously over the connection
- Packet switching divides messages into fixed size units (packets) which are routed through the network individually.
  - different packets in the same message may follow different routes.

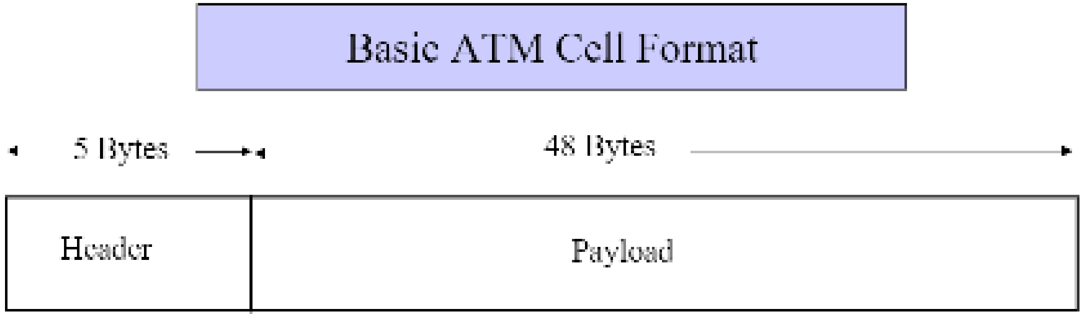
ATM Network (2)

- ATM uses fixed-size packets called "cells." Each 53-byte ATM cell contains ***48 bytes of data payload and 5 bytes of control and routing information in the header.***
- The **header** provides addressing information for switching the packet to its destination.
- The **payload** section carries the actual information, which can be data, voice, or video.
- The payload is properly called the ***user information field.***
- The reason for choosing 48 bytes as the payload size is to compromise between the optimal cell sizes for carrying voice information (32 bytes) and data information (64 bytes).

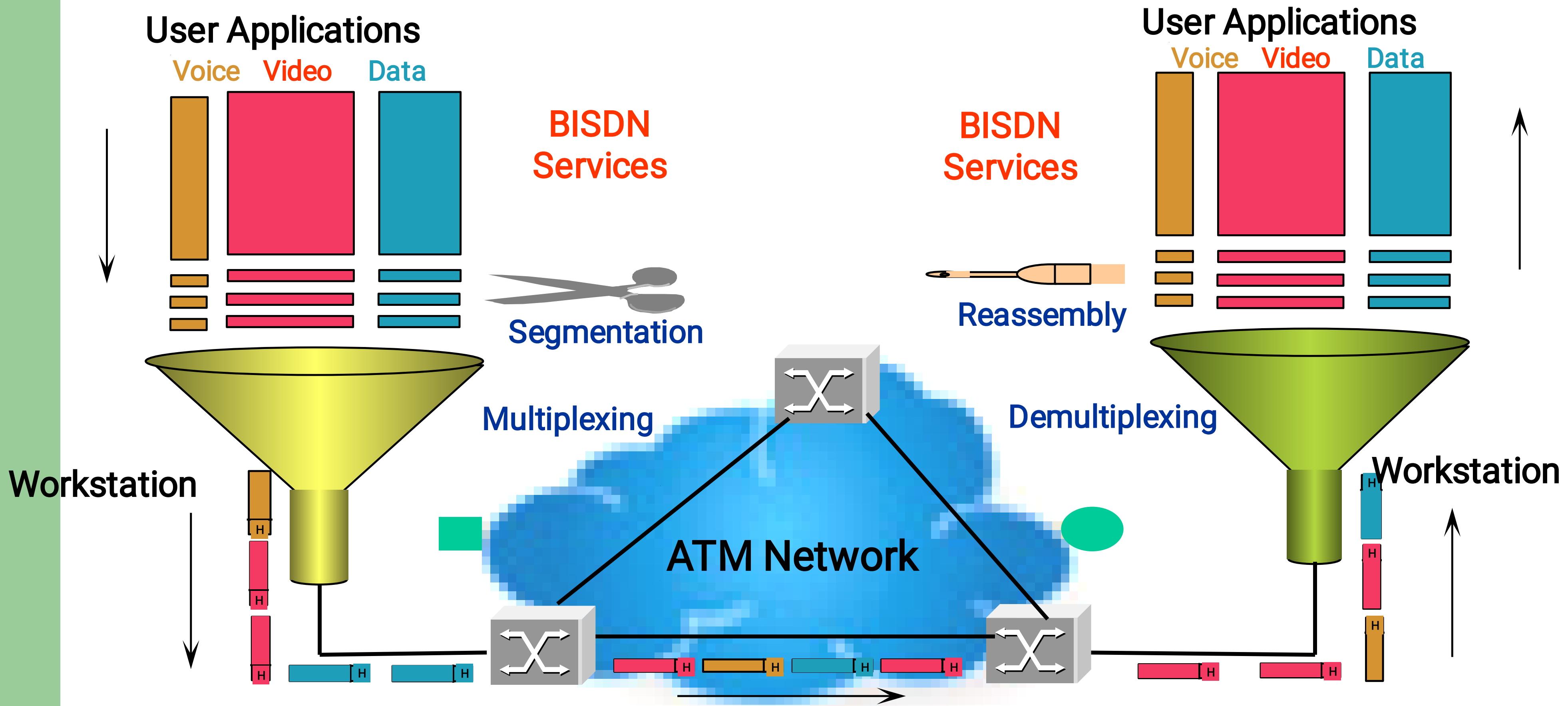
Characteristics of ATM



- Uses small, fixed-sized cells
- Connection-oriented
- Supports multiple service types
- Applicable to LAN and WAN



How Does ATM Work?



ATM Reference Model

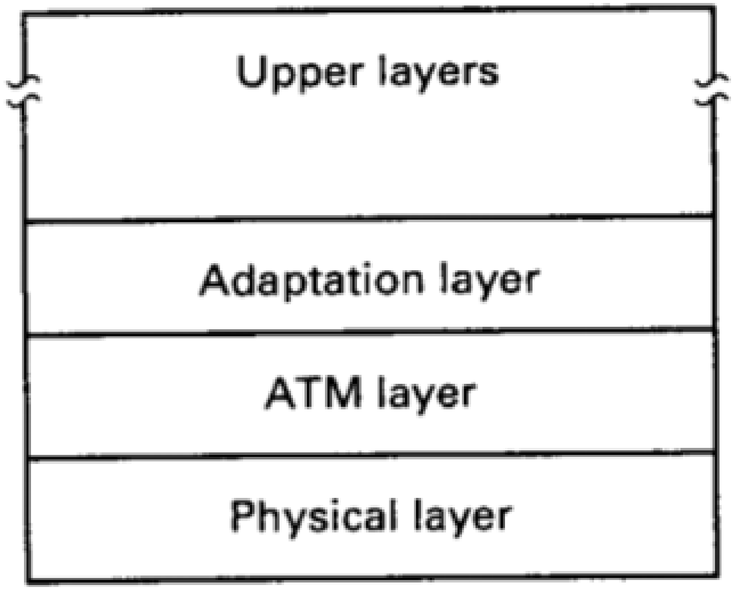


Fig. 2-4. The ATM reference model.

ATM Physical Layer(1)

◆ Designed to use optical technology (SONET)

SONET – Synchronous Optical Network

◆ Essentially digital switch technology

» star topology with switch as central node

» each machine has dedicated connection to switch

» multiple communication paths can be open simultaneously

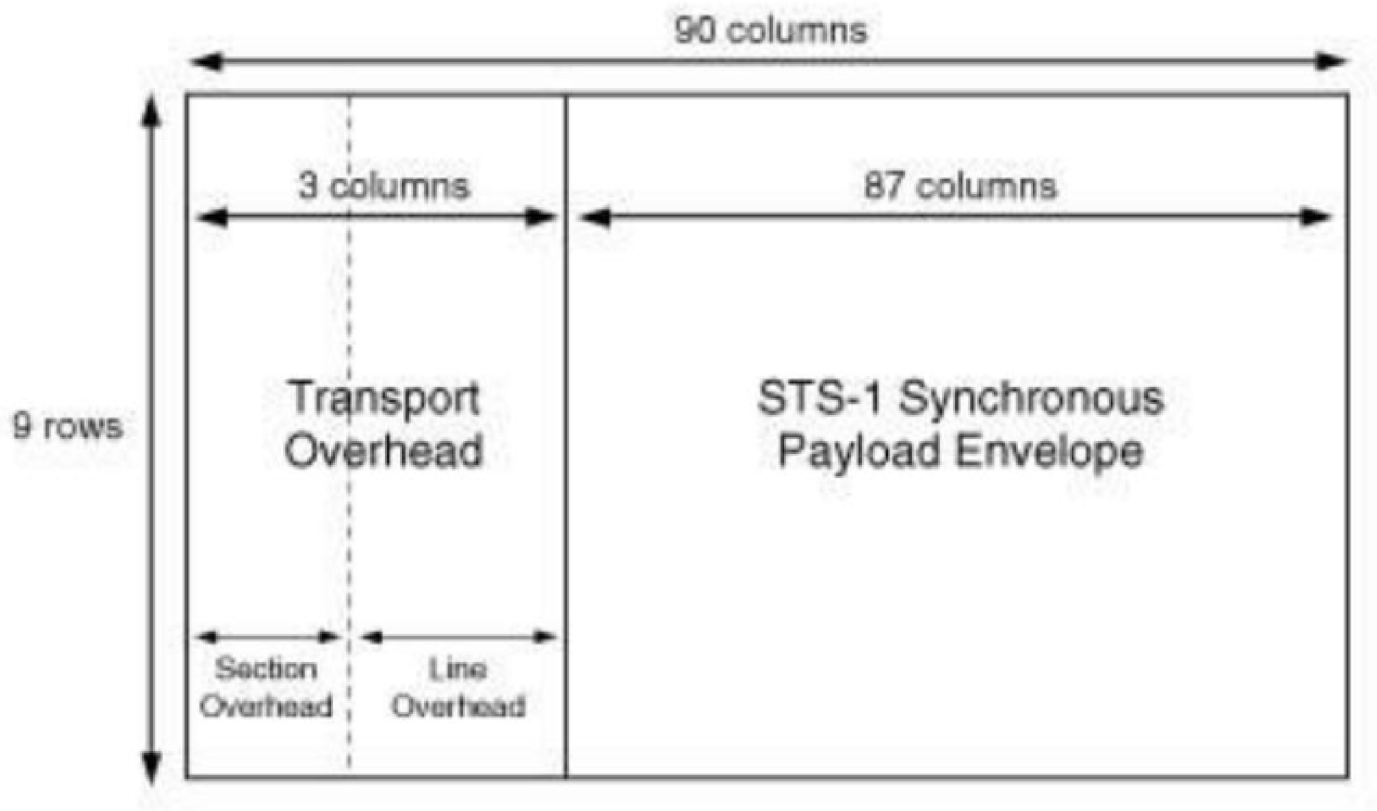
◆ Switching networks...

» allow scaling to large networks

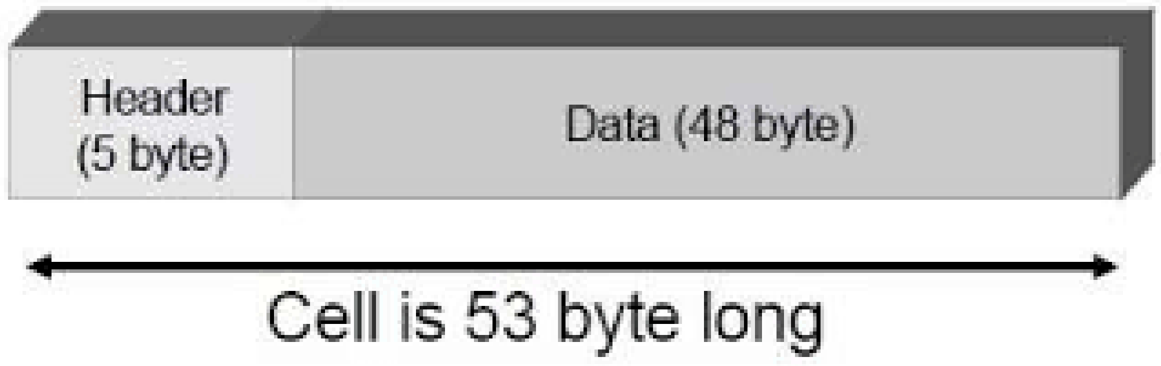
SONET

- SONET uses a basic transmission rate of 51.84 Mbps.
- It is a basic unit of consisting of 9 x 90 array of bytes called a **<u>Frame</u>**.
  - Frame is divided into two areas
    - Transport overhead: First 3 columns (27 bytes)
      - Section overhead (9 bytes)
      - Line overhead (18 bytes)
    - Synchronous Payload Envelope (SPE): Next 87 columns
      - STS Path overhead (9 bytes)
      - Payload (actual message bits)
      - The order of filling data is row-by-row from top-to-bottom and from left-to-right (with MSB first)

SONET

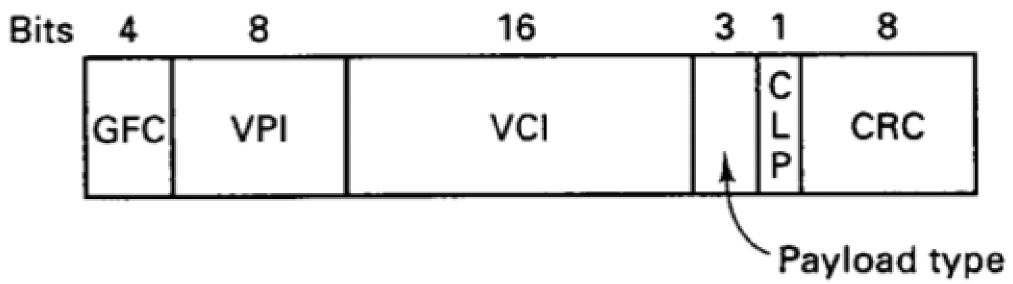

90 columns
3 columns
87 columns
9 rows
Transport Overhead
STS-1 Synchronous Payload Envelope
Section Overhead
Line Overhead

(2)The ATM Layer



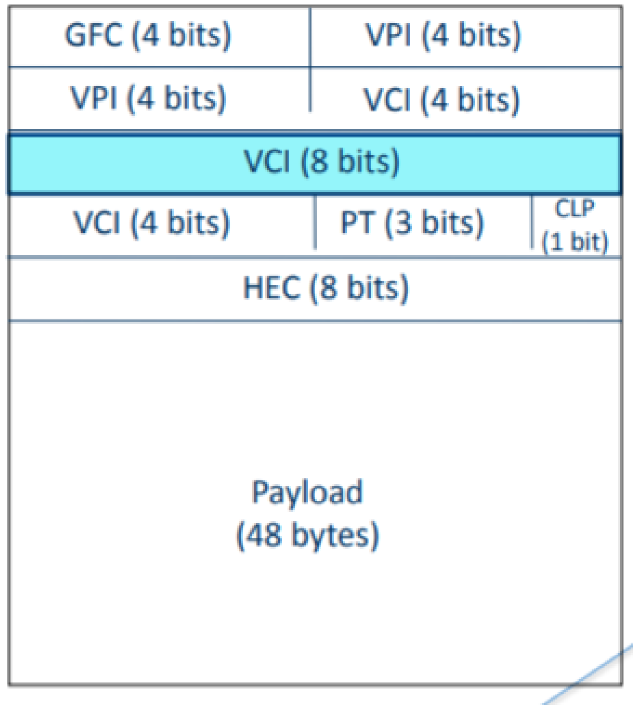
ATM Cell Size

Cell Header Layout (UNI)

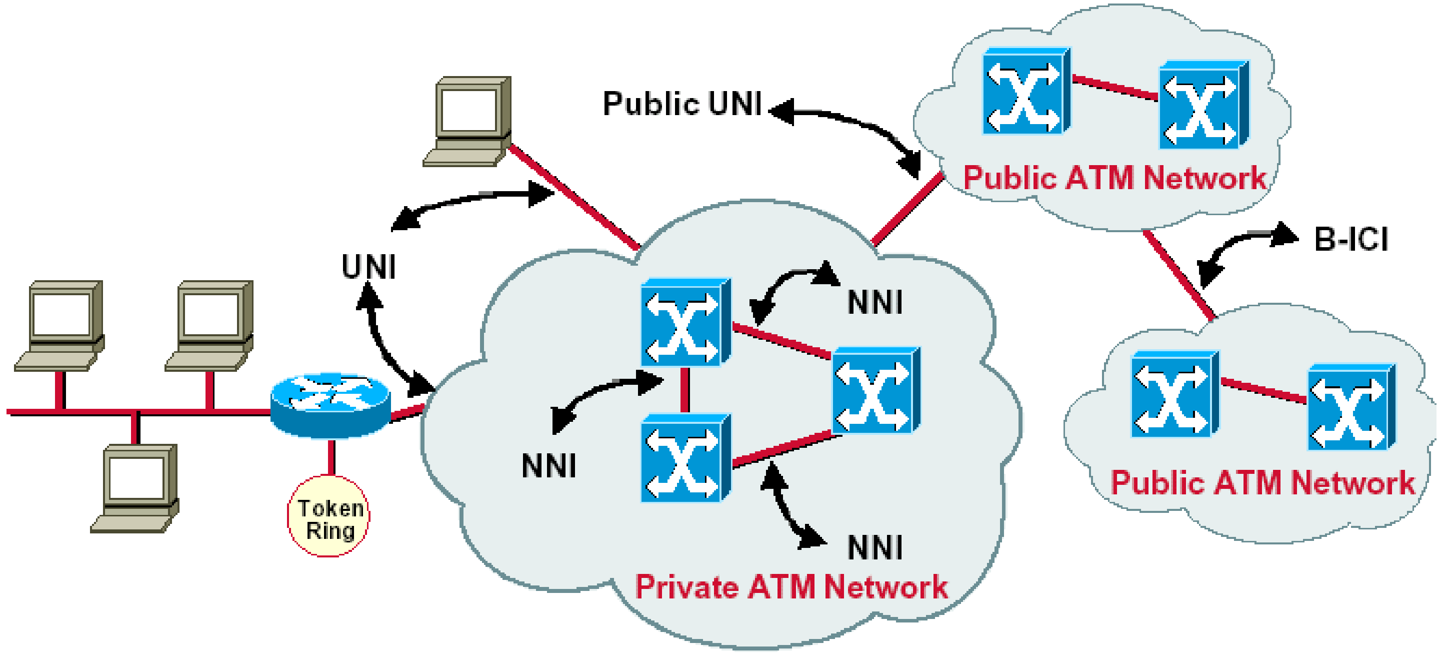


GFC = Generic flow control
VPI = Virtual path idenifier
VCI = Virtual channel identifier
CLP = Cell loss priority
CRC = Cyclic redundancy checksum

Fig. 2-5. User-to-network cell header layout.

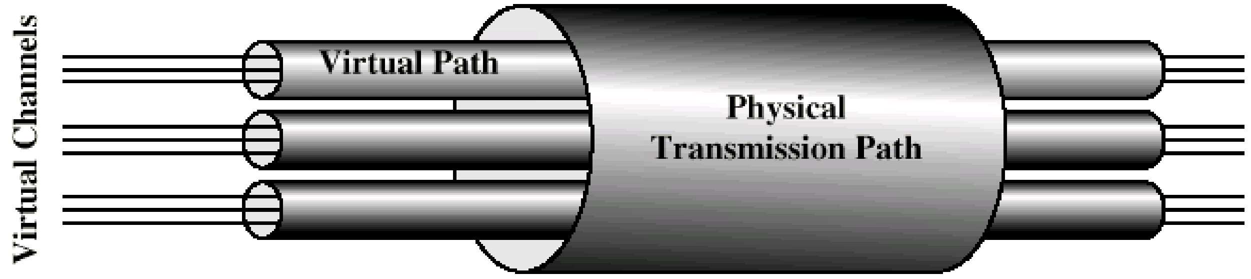


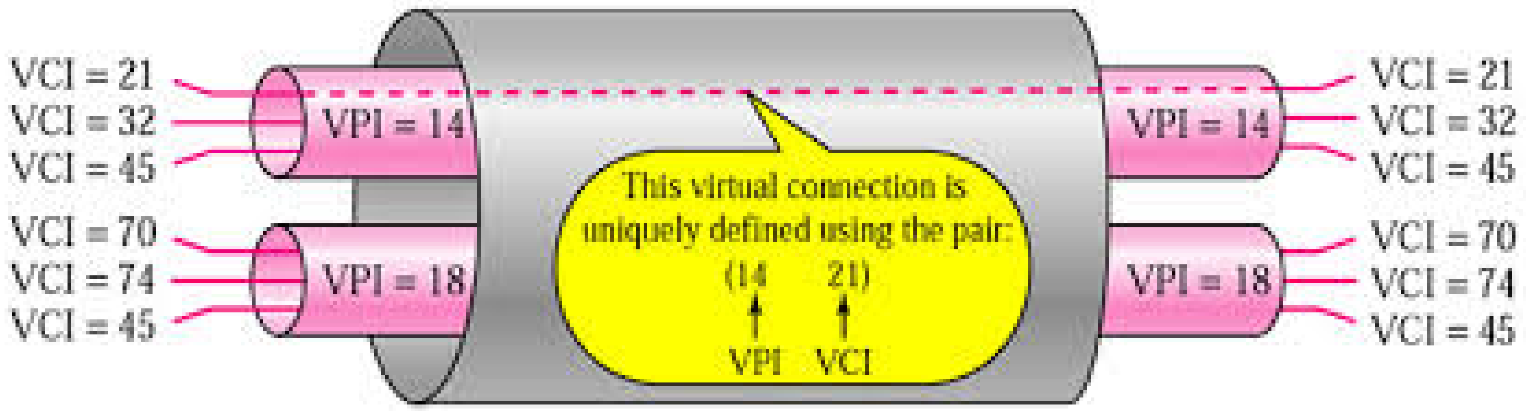
UNI and NNI



- UNI = User-to-Network Interface
- NNI = Network-to-Network Interface

Virtual path vs Virtual Channels





ATM Adaptation Layer

- Originally adaptation layer is designed for four classes of traffic
  A. Constant bit-rate applications (CBR)
  B. Variable bit-rate applications (VBR)
  C. Connection-oriented data applications
  D. Connectionless data application
- Four types
  - Type 1
  - Type 2
  - Type 3/4
  - Type 5

ATM defines four versions of the AAL:

- **AAL1:** Support Constant-bit-rate data (CBR) from upper layer; video and voice.

- **AAL2:** Used for low-bit-rate and short-frame traffic such as audio (compressed or uncompressed), video, or fax. AAL2 allows the multiplexing of short frames into one cell.

- **AAL3/4:** support connection-oriented and connenctionless data services

- **AAL5:** Assumes that all cells belonging to a single message travel sequentially and that control functions are included in the layers of the sending application.

ATM Adaptation Layer (1)

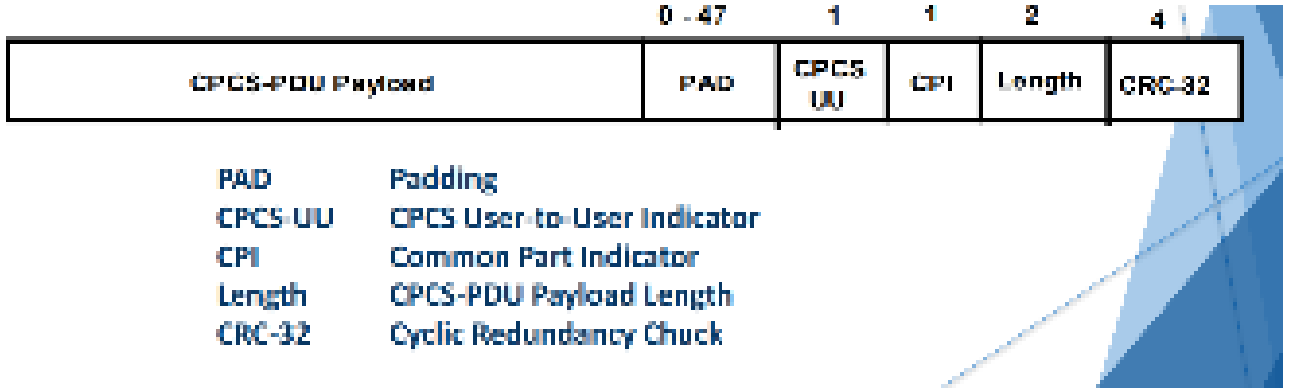
- The AAL interface was initially defined as classes A-D with SAP (service access points) for AAL1-4.
- AAL3 and AAL4 were so similar that they were merged into AAL3/4.
- The data communications community concluded that AAL3/4 was not suitable for data communications applications.
- They pushed for standardization of AAL5 (also referred to as SEAL – the Simple and Efficient Adaptation Layer).
  - AAL2 was not initially deployed

AAL Type 5 Protocol SEAL

- SEAL – Simple and Efficient Adaptation layer.
- The main functions of AAL 5 are segmentation and reassembly. It accepts higher layer packets and segment them into 48-byte ATM cells before transmission via ATM network.
- AAL5 is a simple and efficient AAL (SEAL) to perform a subset of the functions of AAL3/4
- The CPCS-PDU payload length can be up to 65,535 octets and must use PAD (0 to 47 octets) to align CPCS-PDU length to a multiple of 48 octets

SEAL

- Common Part Convergence Sublayer (CPCS)



PAD	Padding
CPCS-UU	CPCS User-to-User Indicator
CPI	Common Part Indicator
Length	CPCS-PDU Payload Length
CRC-32	Cyclic Redundancy Chuck

- The trailer has four fields. The first two are each 1 byte long and are not used, a 2-byte field giving the packet length, and a 4-byte checksum over the packet.

ATM Switching

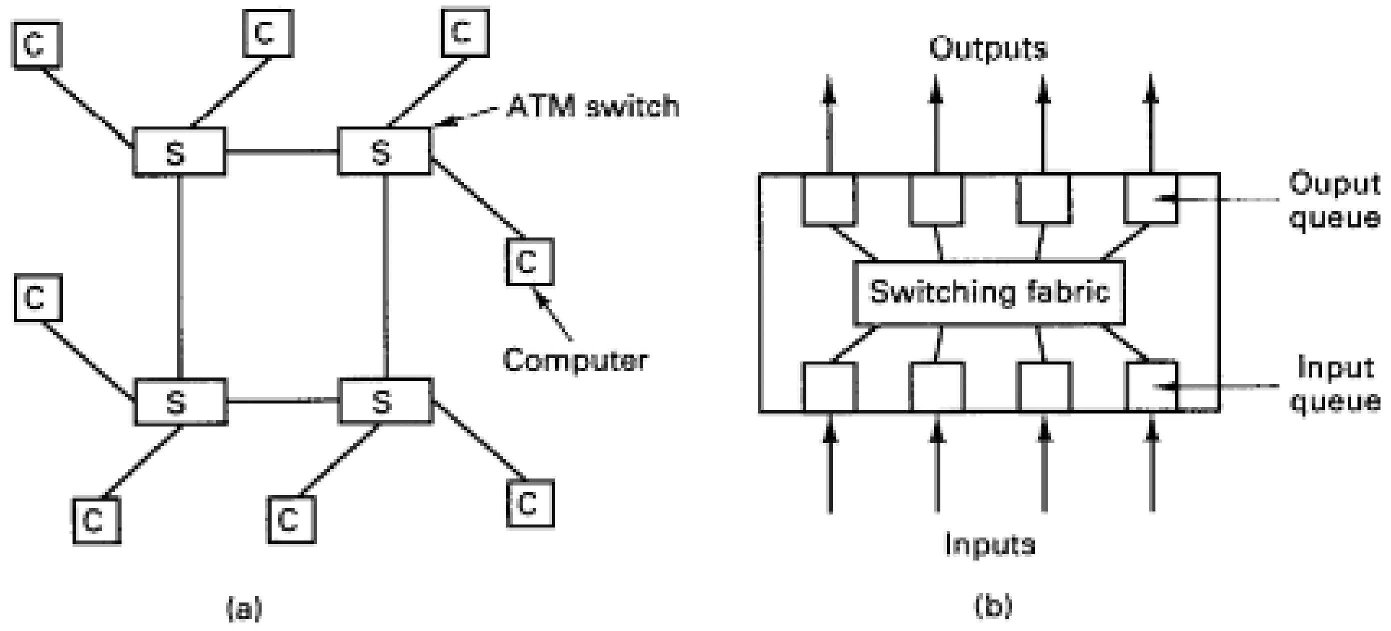


Fig. 2-6. (a) An ATM switching network. (b) Inside one switch.

ATM Switching (2)

- Figure 2-6(a) illustrates a network with four switches.
- Cells originating at any of the eight computers attached to the system can be switched to any of the other computers by traversing one or more switches.
- Each of these switches has four ports, each used for both input and output.

ATM Switching (3)

Inside one switch:

•Has input lines and output lines and a parallel switching fabric that connects them.

•Because a cell has to be switched in 3 secs and as many cells as there are input lines can arrive at once, parallel switching is essential.

Head-of-line Blocking

- **Problem** : When two cells arrive at the same time on different input lines and need to go to the same output port.
- Head-of-line blocking (HOL blocking) in computer networking is a **performance-limiting phenomenon that occurs when a line of packets is held up by the first packet**.
- If two ports, each have stream of cells for the same destination, input queues will build up blocking other cells behind them that want to go to the output ports that are free.
- **SOLUTION:** A different switch design that copies the cell in to a queue associated with the output buffer and lets it wait there, instead of keeping it in the input buffer.

ATM Switching (4)

Other solutions:

- Time division Switches – Using shared memory and buses
- Space division Switches – Having one or more paths between each input and output.

Implications of ATM

- **Some Implications of ATM for Distributed Systems**:
  1) The availability of ATM networks at 155 Mbps, 622 Mbps, and potentially at 2.5 Gbps has some major implications for the design of distributed systems.

<u>Reason:</u> Due to sudden availability of enormously high bandwidth . The effects are most pronounced on wide-area distributed systems.

- Consider sending a 1-Mbit file across the United States and waiting for an acknowledgement that it has arrived correctly.
  - For 1-Mbit - Takes a bit about 15 msec to go across the US.
  - At 64 Kbps, it takes about 15.6 sec
  - As speeds go up, the time-to-reply asymptotically approaches ***30 msec.***
  - For messages shorter than 1 Mbps, which are common in distributed systems, it is even worse.

ATM Switching (5)

The conclusion is: For high-speed wide-area distributed systems, new protocols and system architectures will be needed to deal with the latency in many applications, especially interactive ones.

2) **Flow control:**

A truly large file, say a videotape consisting of 10 GB.

Pblm: **30 msec latency**

CLIENT SERVER MODEL

OSI Model Drawbacks:

• The existence of all those headers generates a considerable amount of overhead.

•Every time a message is sent it must be processed by about ***half a dozen layers***, each one generating and adding a header on the way down or removing and examining a header on the way up.

•On **wide-area networks**, where the number of bits/sec that can be sent is low (often as little as 64K bits/sec), this overhead is not serious.

CLIENT SERVER MODEL

LAN Based Distributed Systems:

•So much CPU time is wasted running protocols.

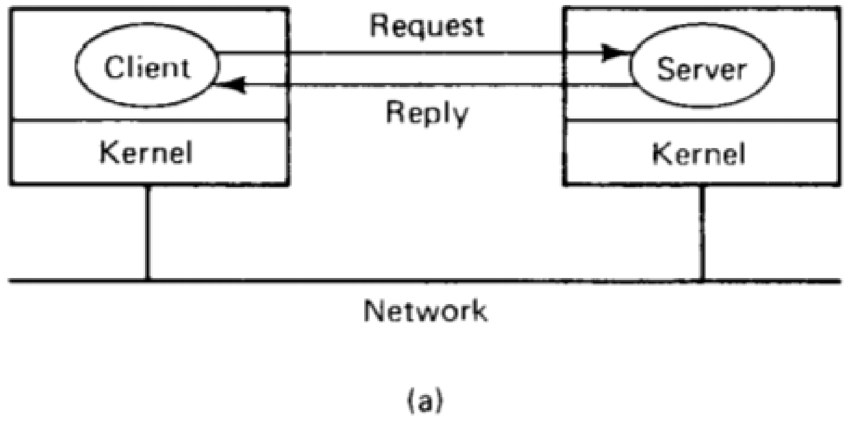
•Use only a subset of the entire protocol stack.

The OSI model addresses only a small aspect of the problem

❑ Getting the bits from the sender to the receiver.

❑ Does not say anything about how the distributed system should be structured.

CLIENT SERVER MODEL



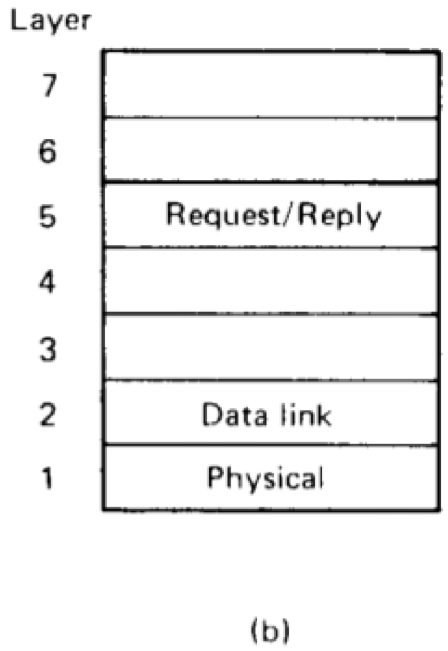


Fig. 2-7. The client-server model. Although all message passing is actually done by the kernels, this simplified form of drawing will be used when there is no ambiguity.

CLIENT SERVER MODEL

- Based on a simple, connectionless request/reply protocol. The client sends a request message to the server asking for some service (e.g., read a block of a file).
- The server does the work and returns the data requested or an error code indicating why the work could not be performed.
- **Advantage – 1) simplicity.**
  - The client sends a request and gets an answer.
  - No connection has to be established before use or torn down afterward.
  - The reply message serves as the acknowledgement to the request.

CLIENT SERVER MODEL

2) **Efficiency:**

The protocol stack is shorter and thus more efficient.

Only **three levels of protocol** are needed.

- The physical and data link protocols take care of getting the packets from client to server and back.
- No routing is needed and no connections are established, so layers 3 and 4 are not needed.
- Layer 5 is the request/reply protocol.
- There is no session management because there are no sessions. The upper layers are not needed either.

CLIENT SERVER MODEL

- Communication services provided by the (micro)kernel can, be reduced to two system calls.
  - One for sending messages and one for receiving them.
  - These system calls can be invoked through library procedures **send(dest, &mptr) and receive(addr, &mptr)**.
  - **Send-** sends the message pointed to by mptr to a process identified by dest and causes the caller to be blocked until the message has been sent.
  - **Receive** - causes the caller to be blocked until a message arrives.

An Example Client and Server

- Client and a file server in C
- Both the client and the server need to share some definitions - collected into a file called ***header.h.***
- Both the client and server include these using the #include statement
  - Has the effect of causing a preprocessor to literally insert the entire contents of header.h into the source program just before the compiler starts compiling the program.

```
/* Definitions needed by clients and servers. */
#define MAX_PATH    255 /* maximum length of a file name */
#define BUF_SIZE   1024 /* how much data to transfer at once */
#define FILE_SERVER 243 /* file server's network address */

/* Definitions of the allowed operations. */
#define CREATE 1 /* create a new file */
#define READ   2 /* read a piece of a file and return it */
#define WRITE  3 /* write a piece of a file */
#define DELETE 4 /* delete an existing file */

/* Error codes. */
#define OK            0 /* operation performed correctly */
#define E_BAD_OPCODE -1 /* unknown operation requested */
#define E_BAD_PARAM  -2 /* error in a parameter */
#define E_IO         -3 /* disk error or other I/O error */

/* Definition of the message format. */
struct message {
 long source; /* sender's identity */
 long dest; /* receiver's identity */
 long opcode; /* which operation: CREATE, READ, etc. */
 long count; /* how many bytes to transfer */
 long offset; /* where in file to start reading or writing */
 long extra1; /* extra field */
 long extra2; /* extra field */
 long result; /* result of the operation reported here */
 char name[MAX_PATH]; /* name of the file being operated on */
 char data[BUF_SIZE]; /* data to be read or written */
};
```
Fig. 2-8. The *header.h* file used by the client and server.

```
#include
void main(void) {
 struct message m1, m2; /* incoming and outgoing messages */
 int r; /* result code */
 while (1) { /* server runs forever */
  receive(FILE_SERVER,&m1); /* block waiting for a message */
  switch(m1.opcode) { /* dispatch on type of request */
  case CREATE: r = do_create(&m1, &m2); break;
  case READ: r = do_read(&m1, &m2); break;
  case WRITE: r = do_write(&m1, &m2); break;
  case DELETE: r = do_delete(&m1, &m2); break;
  default: r = E_BAD_OPCODE;
 }
 m2.result = r; /* return result to client */
 send(m1.source, &m2); /* send reply */
 }
}
```
(a)

```
#include
int copy(char *src, char *dst) /* procedure to copy file using the server */
{
 struct message m1; /* message buffer */
 long position; /* current file position */
 long client = 110; /* client's address */
 initialize(); /* prepare for execution */
 position = 0;
 do {
  /* Get a block of data from the source file. */
  m1.opcode = READ; /* operation is a read */
  m1.offset = position; /* current position in the file */
  m1.count = BUF_SIZE; /* how many bytes to read */
  strcpy(&m1.name, src); /* copy name of file to be read to message */
  send(FILE_SERVER, &m1); /* send the message to the file server */
  receive(client, &m1); /* block waiting for the reply */
  /* Write the data just received to the destination file. */
  m1.opcode = WRITE; /* operation is a write */
  m1.offset = position; /* current position in the file */
  m1.count = m1.result; /* how many bytes to write */
  strcpy(&m1.name, dst); /* copy name of file to be written to buf */
  send(FILE_SERVER, &m1); /* send the message to the file server */
  receive(client, &m1); /* block waiting for the reply */
  position += m1.result; /* m1.result is number of bytes written */
 } while (m1.result > 0); /* iterate until done */
 return(m1.result >= 0 ? OK : m1.result); /* return OK or error code */
}
```
(b)

Fig. 2-9. (a) A sample server. (b) A client procedure using that server to copy a file.

Addressing

- For a client to send a message to a server - Must know the server's address.
- In the example of the preceding section, the server's address was simply hardwired into header.h as a constant.
- This strategy might work in a simple system - more sophisticated form of addressing is needed.

Addressing (2)

- File server has been assigned a numerical address (243), but - Not really specified what this means.
  - Does it refer to a *specific machine, or to a specific process*?
- Sending kernel can extract it from the message structure and use it as the hardware address for sending the packet to the server.
  - Build a frame using the 243 as the data link address and put the frame out on the LAN - Server's interface board see the frame, recognize 243 as its own address, and accept it.

Addressing (3)

- If only one process on destination machine, the kernel will give it to the one and only process running there.
- If there are several processes running on the destination machine, The kernel has no way of knowing which one gets the message? .
- *Only one process can run on each machine, which is a serious restriction*

Addressing (4)

(a) ***Alternative addressing system: [machine.process addressing]***

- Sends messages to processes rather than to machines.

Problem: How processes are identified.

- One common scheme is to use two part names, specifying both a machine and a process number.
- Thus 243.4 or 4@243
  - **Machine number** - used by the kernel to get the message correctly delivered to the proper machine
  - **Process number** - used by the kernel on that machine to determine which process the message is intended for.
- Advantage of this approach: Every machine can number its processes starting at 0. No global coordination is needed because there is never any ambiguity between process 0 on machine 243 and process 0 on machine 199. The former is 243.0 and the latter is 199.0. This scheme is illustrated in Fig. 2-10(a)

Addressing (5)

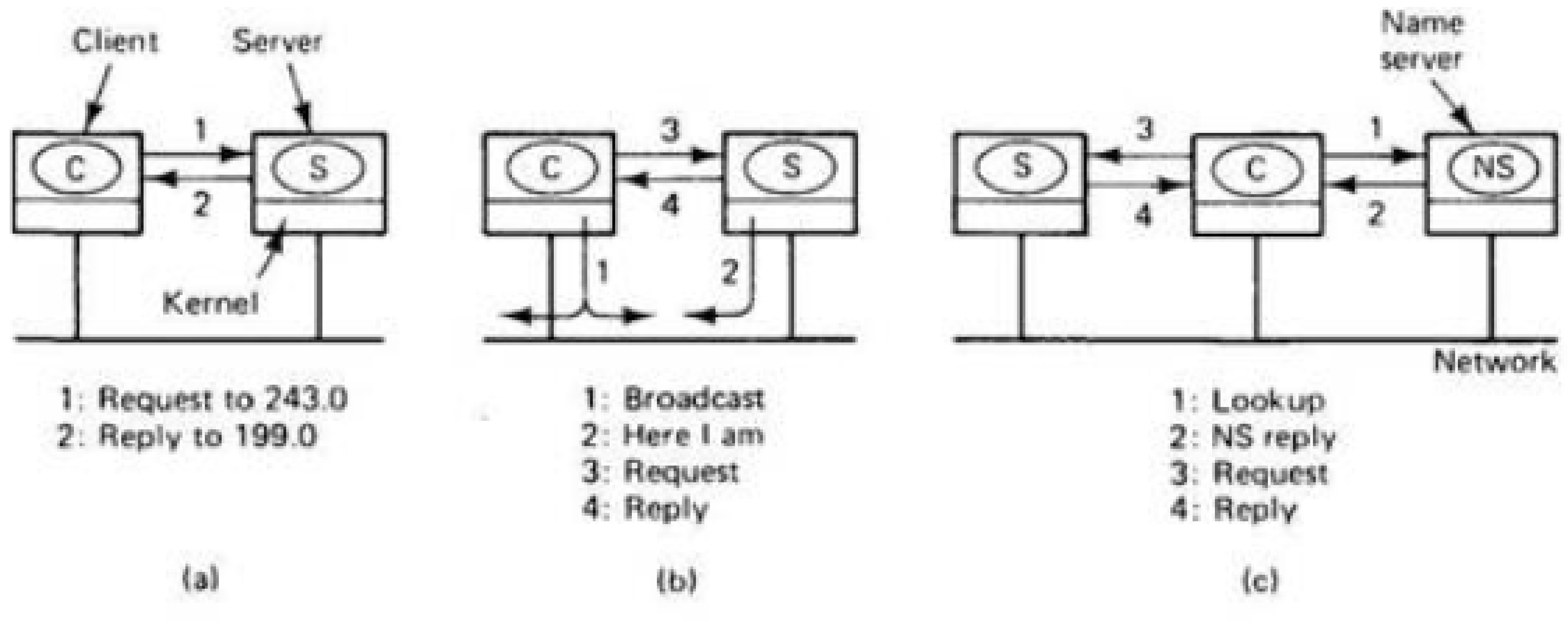


Fig. 2-10. (a) Machine.process addressing. (b) Process addressing with broadcasting. (c) Address lookup via a name server.

Addressing (6)

(b) Process addressing with broadcasting

- Machine.process addressing is not transparent since the user is obviously aware of where the server is located, and transparency is one of the main goals of building a distributed system.
  - Eg: Suppose that the file server normally runs on machine 243, but if that machine is down and Machine 176 is available, but programs previously compiled using header.h all have the number 243 built into them, so they will not work if the server is unavailable.
  - Clearly, this situation is undesirable.

Addressing (7)

An alternative approach:

- To assign each process a unique address that *does not contain an embedded machine number.*
  - To achieve this, a **centralized process address allocator** that simply maintains a **counter** can be used.
- Upon receiving a request for an address, it simply returns the current value of the counter and then **increments it by one**.
- **Disadvantage:** Centralized components like this do not scale to large systems and thus should be avoided.

Addressing (8)

Another method for assigning process identifiers:

- To let each process pick its own identifier from a large, sparse address space, such as the space of 64-bit binary integers.
- The probability of two processes picking the same number is tiny, and the system scales well.
- **Problem:** How does the sending kernel know what machine to send the message to?
- On a LAN that supports broadcasting, the sender can broadcast a special locate packet containing the address of the destination process. This method is shown in Fig. 2-10(b).

Basic primitives

1. Send
2. Receive

- Send has two parameters
  - Message and a destination
- Receive has two parameters
  - A source and a buffer

Blocking and Nonblocking Primitives

SEND - Blocking Primitives

A CALL TO SEND:

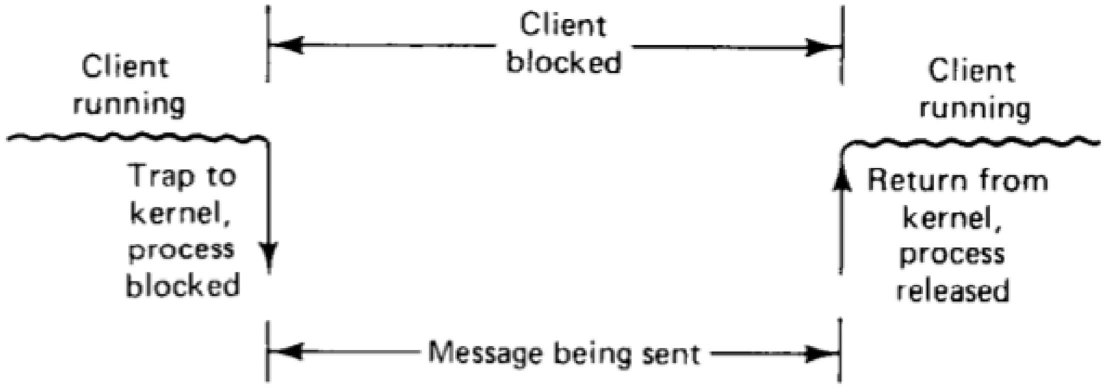
•When a process calls send it specifies ***a destination*** and ***a buffer*** to send to that destination.

•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 completely sent, as shown in Fig. 2-1l(a).

SEND - Blocking Primitives (1)

- Fig. 2-11. (a) A blocking send primitive.



Client blocked
Client running
Client running
Trap to kernel, process blocked
Return from kernel, process released
Message being sent

(a)

RECEIVE - Blocking Primitives (2)

A CALL TO RECEIVE:

• A call to receive - Does not return control until a message has actually been received and put in the message buffer pointed to by the parameter.

•The process remains suspended in receive until a message arrives, even if it takes hours.

SEND - Nonblocking primitives

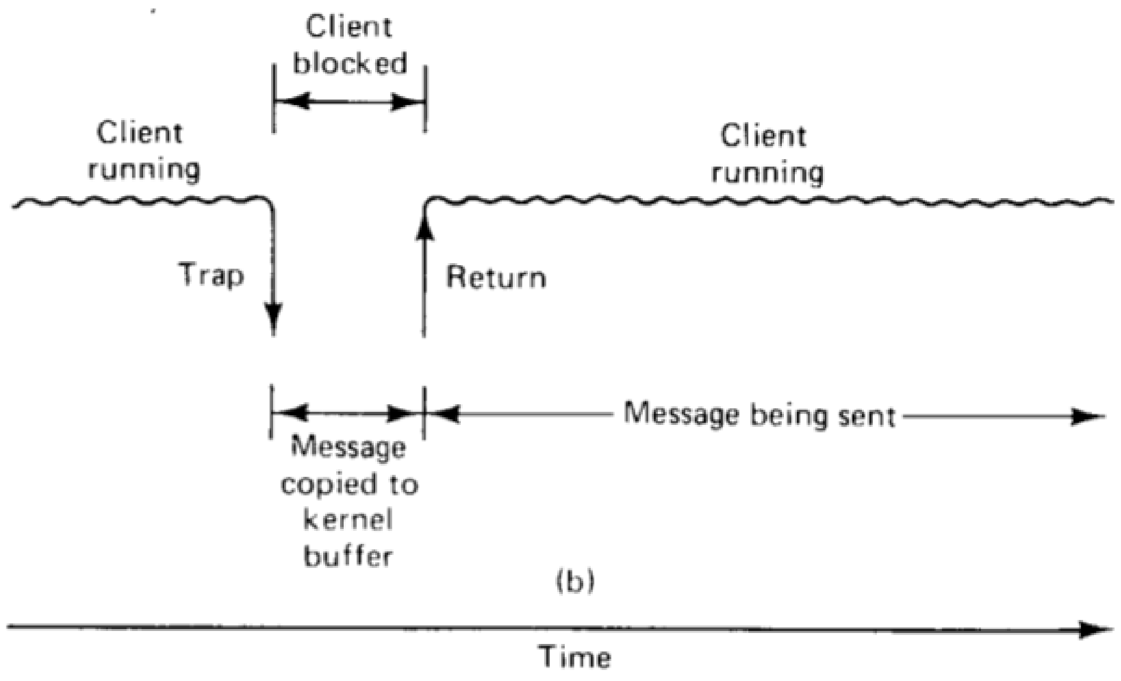
- Alternative to blocking primitives(Also called asynchronous primitives).

SEND:

- If send is nonblocking, it returns control to the caller immediately, before the message is sent.
- **Advantage:** Sending process can continue computing in parallel with the message transmission, instead of having the CPU go idle (assuming no other process is runnable).
- The choice - Made by the system designers.

SEND - Nonblocking primitives (1)

- (b) A nonblocking send primitive.



SEND - Nonblocking primitives (2)

Disadvantage:

•Sender cannot modify the message buffer until the message has been sent.

•Sending process has no idea of when the transmission is done, so it never knows when it is safe to reuse the buffer.

SEND - Nonblocking primitives (3)

- **SOLUTIONS**
  - SOLUTION (1):
- To have the kernel copy the message to an internal kernel buffer and then allow the process to continue.
- **<u>ADV</u>** - From the sender's point of view, this scheme is the same as a blocking call. Of course, the message will not yet have been sent, but the sender is not hindered by this fact.
- **<u>DISADV</u>**: Every outgoing message has to be copied from user space to kernel space.

SEND - Nonblocking primitives (4)

SOLUTION (2):

- To interrupt the sender when the message has been sent to inform it that the buffer is once again available.
- ADV:
  - No copy is required here, which saves time.
  - Highly efficient and allows the most parallelism.
- DISADV:
  - Programs based on interrupts are difficult to write correctly
  - Nearly impossible to debug when they are wrong.

THREAD OF CONTROL

- If only a single thread of control is available, the choices come down to:
  - 1. Blocking send (CPU idle during message transmission).
  - 2. Nonblocking send with copy (CPU time wasted for the extra copy).
  - 3. Nonblocking send with interrupt (makes programming difficult).

SEND - Nonblocking primitives (5)

- CONCLUSION:
  - The difference between a synchronous primitive and an asynchronous one is whether the sender can reuse the message buffer immediately after getting control back without fear of messing up the send. When the message actually gets to the receiver is irrelevant.

RECEIVE - Nonblocking primitives

RECEIVE:

- A nonblocking receive just tells the kernel where the buffer is, and returns control almost immediately.
- Again here, how does the caller know when the operation has completed?
- **<u>Solution:</u>**
  - To provide an explicit wait primitive that allows the receiver to block when it wants to.
  - To provide a test primitive to allow the receiver to poll the kernel to check on the status.

RECEIVE - Nonblocking primitives

Conditional_receive:

•Either gets a message or signals failure, but in any event returns immediately, or within some timeout interval.

• ***Interrupts*** can also be used to ***signal completion***.

•For the most part, a blocking version of receive is much simpler and greatly preferred.

Issues

- Timeouts:
  - In a system in which send calls block, if there is no reply, the sender will block forever.
  - To prevent this situation, in some systems the caller may specify a time interval within which it expects a reply.
  - If none arrives in that interval, the send call terminates with an error status.

Buffered versus Unbuffered Primitives

Buffered Primitives

Buffered - An address refers to a specific process.

•A call receive(addr, &m) tells the kernel of the machine on which it is running that the calling process is listening to 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.

Unbuffered Message Passing

- From one user buffer to another user buffer directly
- Program using send should avoid reusing the buffer until the message has been transmitted
- For large systems a combination of unbuffered and non blocking semantics allows almost complete overlap between the communication and the on going computational activity in the user program

Buffered Message Passing

1. From user buffer to kernel buffer
2. From the kernel on the sending computer to the kernel buffer on the receiving computer
3. Finally from the buffer on the receiving computer to a user buffer

Buffered versus Unbuffered Primitives

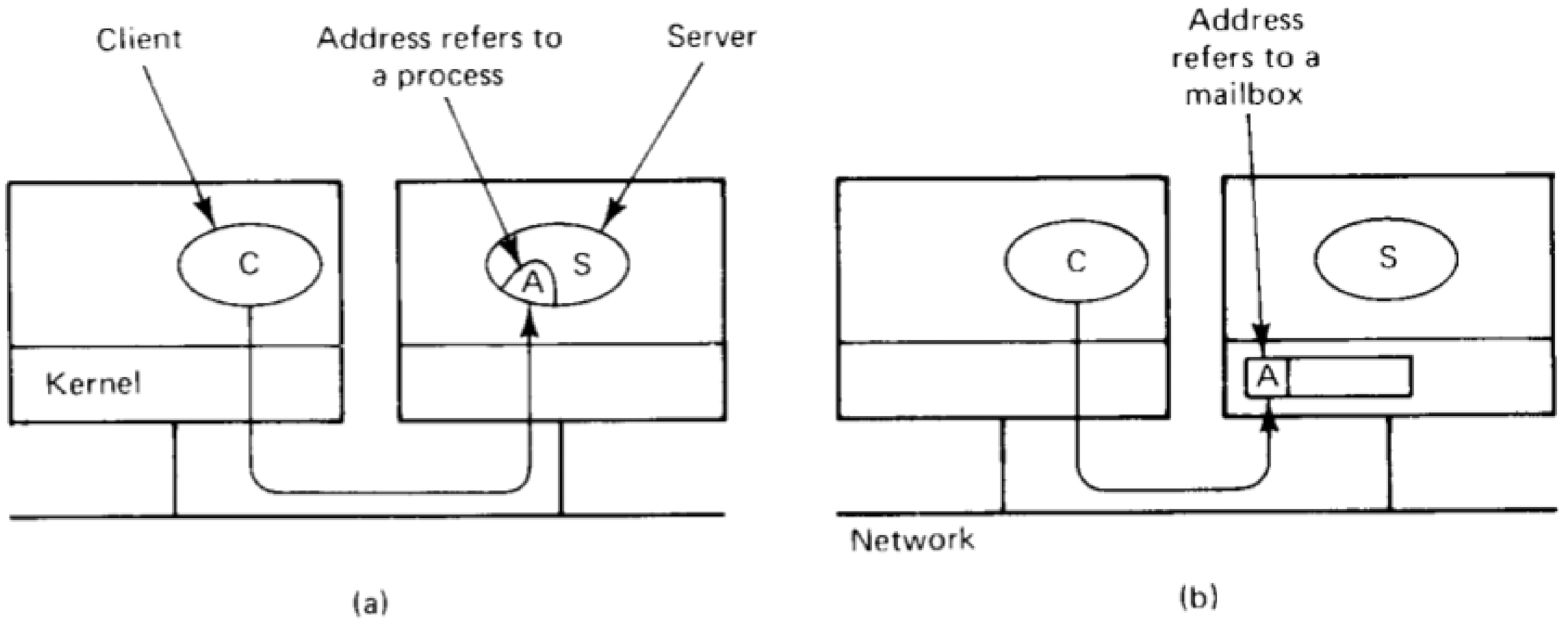


Fig. 2-12. (a) Unbuffered message passing. (b) Buffered message passing.

Buffered versus Unbuffered Primitives

Approach (1):

•To just discard the message, let the client time out, and hope the server has called receive before the client retransmits.

DISADV:

❑Easy to implement - but client to try several times before succeeding - It may give up, falsely concluding that the server has crashed or that the address is invalid.

❑If two or more clients are using the server – Condition is worse even.

Buffered versus Unbuffered Primitives

- **Approach (2):**

To have the receiving kernel keep incoming messages around for a little while.

ADV: Reduces the chance that a message will have to be thrown away.

DISADV: Introduces the problem of storing and managing prematurely arriving messages. Buffers are needed and have to be allocated, freed, and generally managed.

Buffered versus Unbuffered Primitives

- MAILBOX CONCEPT:
  - A process that is interested in receiving messages tells the kernel to create a mailbox for it, and specifies an address to look for in network packets.
  - All incoming messages with that address are put in the mailbox.
  - The call to receive now just removes one message from the mailbox, or blocks (assuming blocking primitives) 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 frequently referred to as a buffered primitive

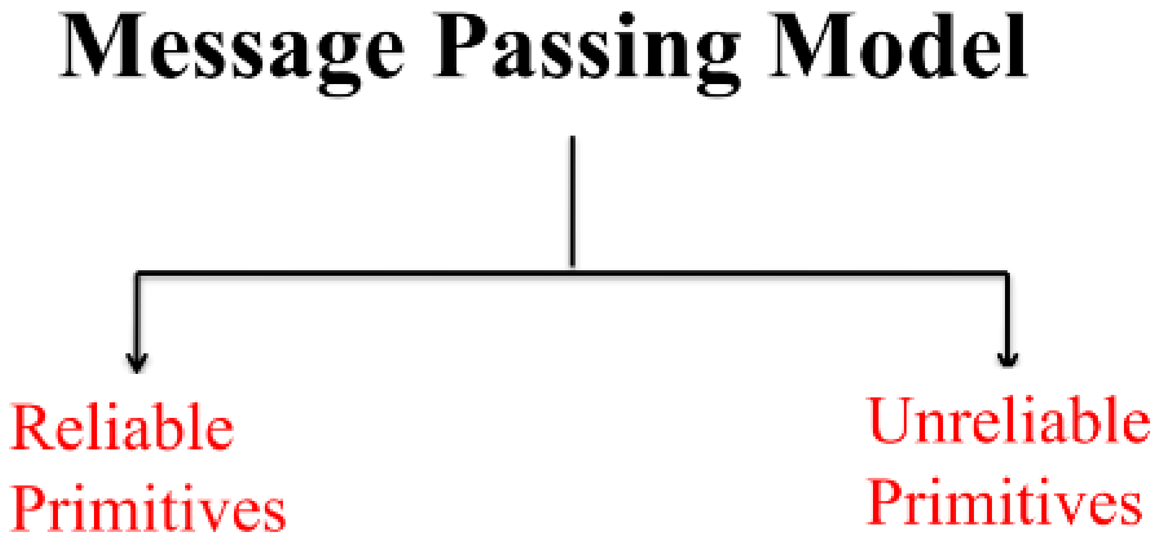
Buffered versus Unbuffered Primitives

- ***Another option:***
  - Do not let a process send a message if there is no room to store it at the destination.

Reliable versus Unreliable

Reliable versus Unreliable Primitives

- So far, when a client sends a message, the server will receive it.
- Messages can get lost, which affects the semantics.



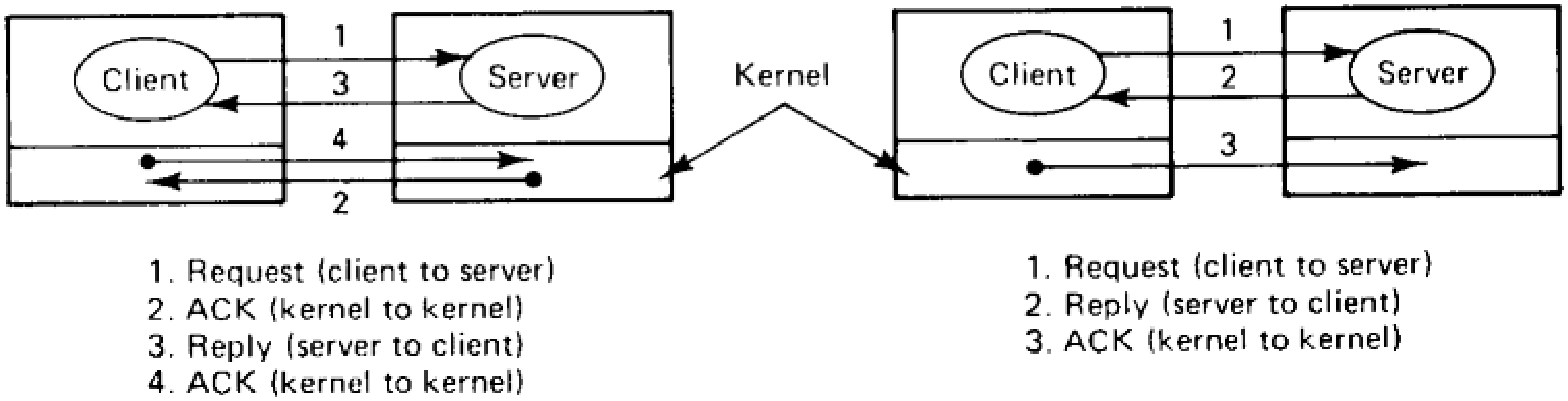
Reliable versus Unreliable Primitives

- Suppose that blocking primitives are being used.
- When a client sends a message, it is suspended until the message has been sent.
- No guarantee that the message has been delivered.
- *Three different approaches* to this problem are possible.

Reliable versus Unreliable Primitives

- Approach (1):
  - To redefine the semantics of send to be unreliable.
  - The system gives no guarantee about messages being delivered.
  - EG: POST OFFICE
- Approach (2):
  - To require the kernel to send an acknowledgement back to the kernel on the sending machine.
  - A request and reply take four messages.

Reliable versus Unreliable Primitives



Client
Server
1
3
4
2
Kernel
Client
Server
1
2
3
1. Request (client to server)
2. ACK (kernel to kernel)
3. Reply (server to client)
4. ACK (kernel to kernel)
1. Request (client to server)
2. Reply (server to client)
3. ACK (kernel to kernel)

Fig. 2-13. (a) Individually acknowledged messages. (b) Reply being used as the acknowledgement of the request. Note that the ACKs are handled entirely within the kernels.

Reliable versus Unreliable Primitives

- Approach (3):
  - To take advantage of the fact that client-server communication is structured as a request from the client to the server followed by a reply from the server to the client.

Reliable versus Unreliable (SUMMARY)

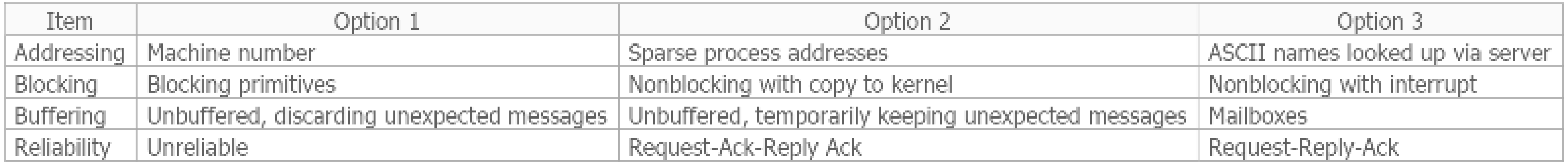
- Unreliable SEND : It does not return control to the user program until the message has been sent.
- Reliable SEND : It does not return control to the user program until an acknowledgment has been received.
- RECEIVE : does not return control until a message is copied to the user buffer.

 Reliable RECEIVE automatically sends an acknowledgment.

 Unreliable RECEIVE does not send an acknowledgment.

Implementing the Client-Server Model

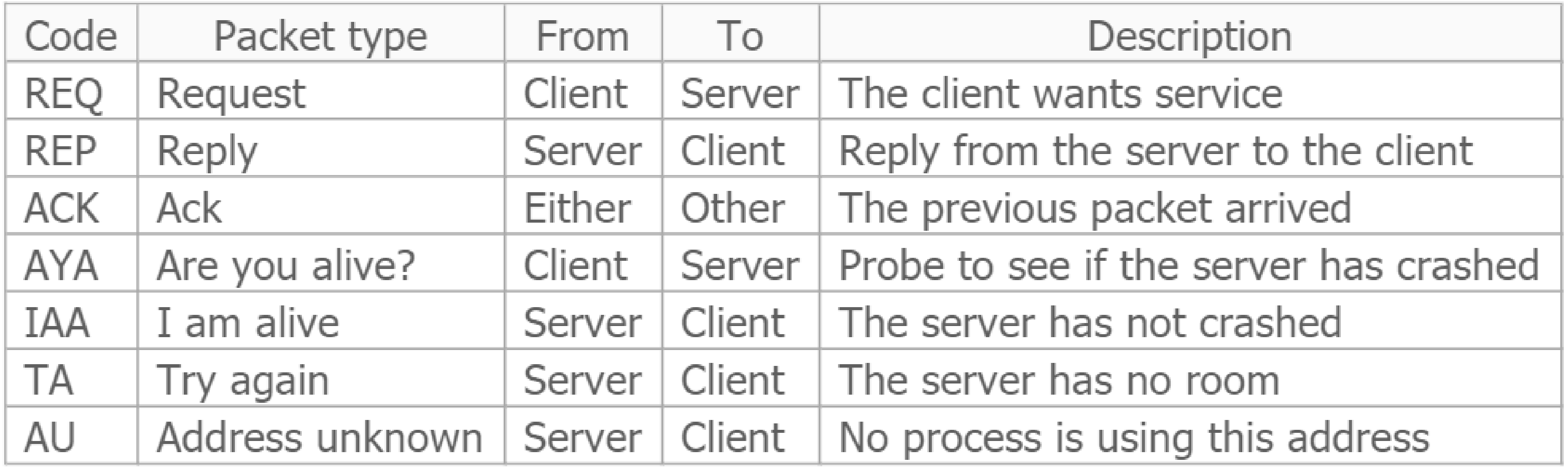
- Four design issues:
  - Addressing,
  - Blocking,
  - Buffering, and
  - Reliability



Item	Option 1	Option 2	Option 3
Addressing	Machine number	Sparse process addresses	ASCII names looked up via server
Blocking	Blocking primitives	Nonblocking with copy to kernel	Nonblocking with interrupt
Buffering	Unbuffered, discarding unexpected messages	Unbuffered, temporarily keeping unexpected messages	Mailboxes
Reliability	Unreliable	Request-Ack-Reply Ack	Request-Reply-Ack

Fig. 2-14. Four design issues for the communication primitives and some of the principal choices available.

Implementing the Client-Server Model



Code	Packet type	From	To	Description
REQ	Request	Client	Server	The client wants service
REP	Reply	Server	Client	Reply from the server to the client
ACK	Ack	Either	Other	The previous packet arrived
AYA	Are you alive?	Client	Server	Probe to see if the server has crashed
IAA	I am alive	Server	Client	The server has not crashed
TA	Try again	Server	Client	The server has no room
AU	Address unknown	Server	Client	No process is using this address

Fig. 2-15. Packet types used in client-server protocols.

Implementing the Client-Server Model

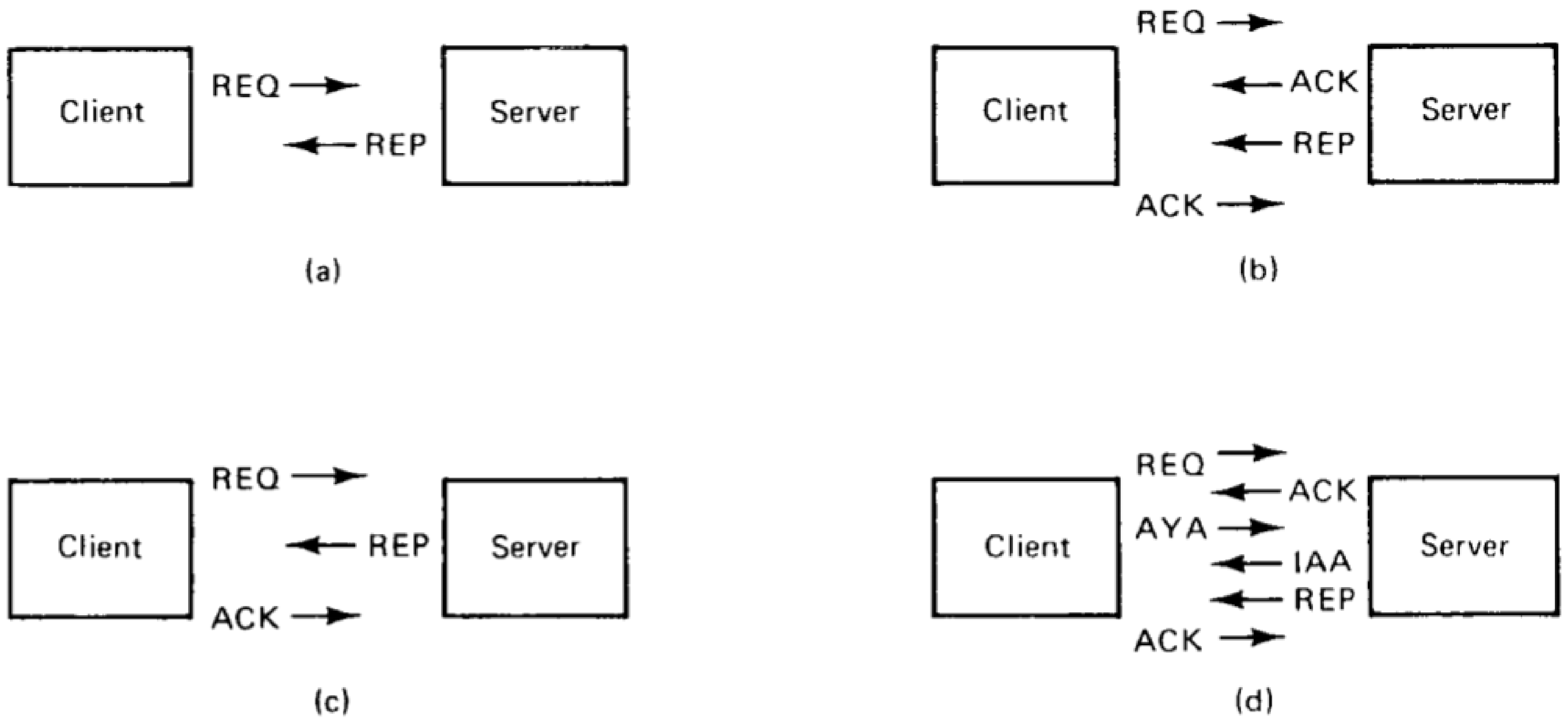


Fig. 2-16. Some examples of packet exchanges for client-server communication.

Remote Procedure Call
RPC

RPC

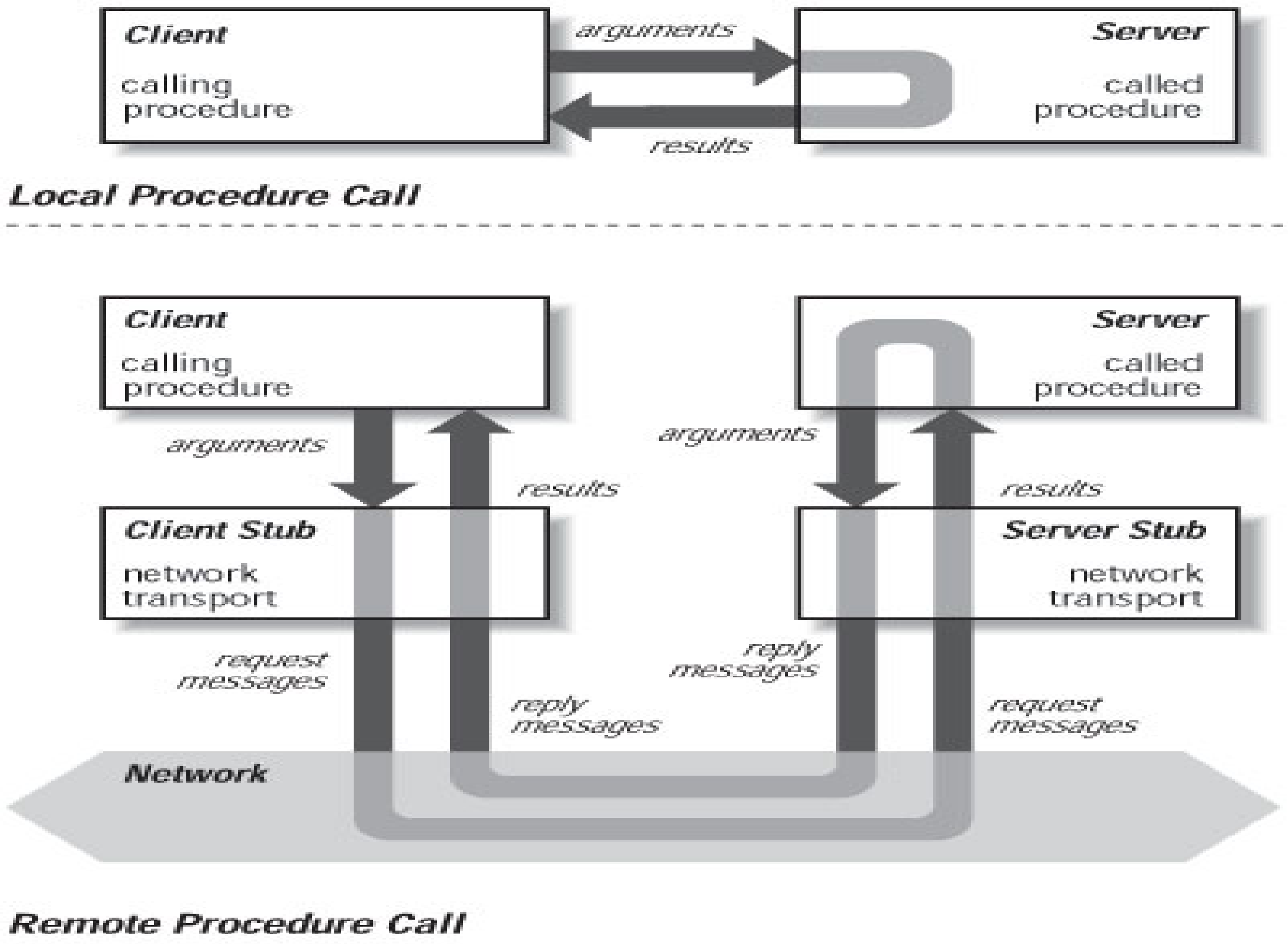
1984: Birrell & Nelson

- Mechanism to call procedures on other machines

Remote Procedure Call

Goal: it should appear to the programmer that a normal call is taking place

Local vs. Remote Procedure Calls



In a local procedure call, a calling process executes a procedure in its own address space.

In a remote procedure call, the client and server run as two separate processes. It is not necessary for them to run on the same machine.

The two processes communicate through stubs, one each for the client and server. These stubs are pieces of code that contain functions to map local procedure calls into a series of network RPC function calls.

Remote Procedure Calls

- Goal: Make distributed computing look like centralized computing
  - Aims at hiding most of the intricacies of message passing, and ideal for client-server applications
- Allow remote services to be called as procedures
  - Transparency with regard to location, implementation, language

Remote procedure call

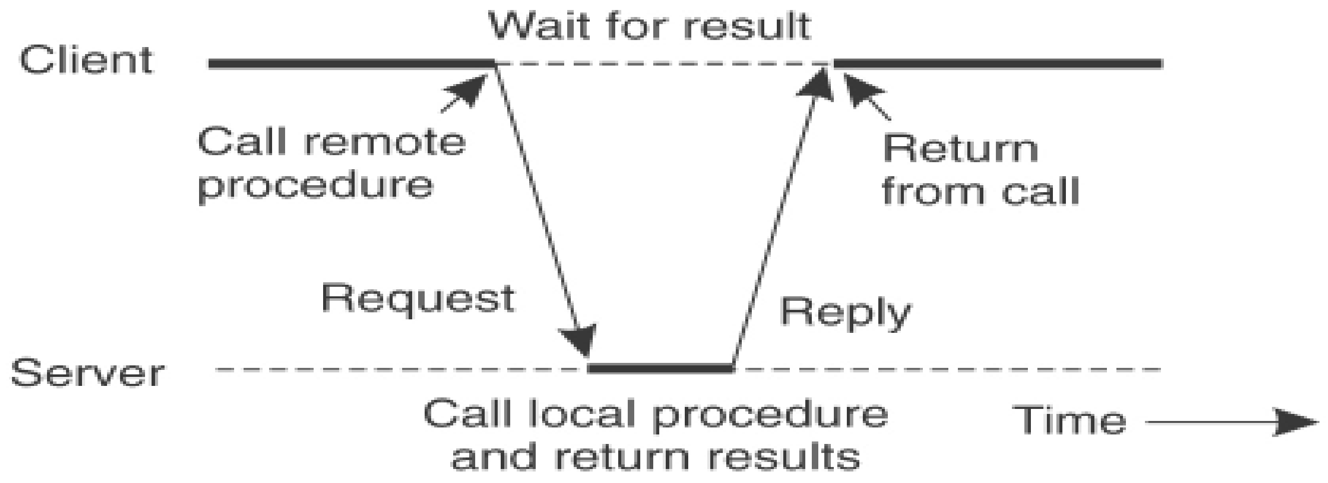
- A remote procedure call makes a call to a remote service look like a local call
  - RPC makes transparent whether server is local or remote
  - RPC allows applications to become distributed transparently
  - RPC makes architecture of remote machine transparent

Possible Issues

- Calling and called procedures run on different machines
- They execute in different address spaces
- Parameters and results have to be passed, it can be complicated when the machines are not identical.
  - How do you represent integers – big-endian little-endian
- Either or both machines can crash and each of the possible failures causes different problems.

Client and Server Stub

- Would like to do the same if called procedure or function is on a remote server



Principle of RPC between a client and server program.

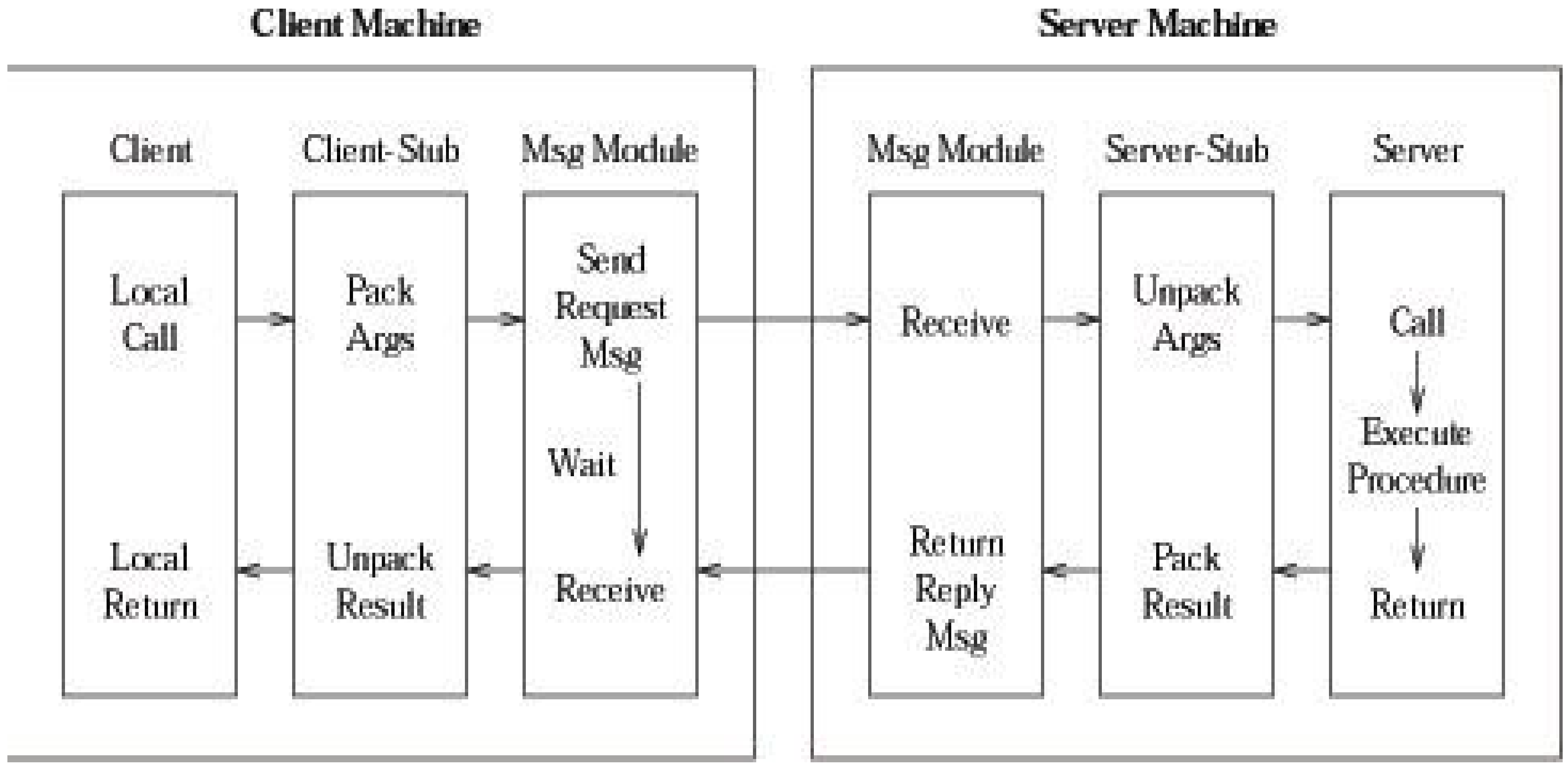
Solution — a pair of *Stubs*

- Client-side stub
  - Looks like local server function
  - Same interface as local function
  - Bundles arguments into message, sends to server-side stub
  - Waits for reply, un-bundles results
  - returns
- Server-side stub
  - Looks like local client function to server
  - Listens on a socket for message from client stub
  - Un-bundles arguments to local variables
  - Makes a local function call to server
  - Bundles result into reply message to client stub

- **Client stub**
  - packs the arguments with the procedure name or ID into a message
  - sends the msg to the server and then awaits a reply msg
  - unpacks the results and returns them to the client

- **Server stub**
  - receives a request msg
  - unpacks the arguments and calls the appropriate server procedure
  - when it returns, packs the result and sends a reply msg back to the client

RPC System Components and Call Flows



Steps of a Remote Procedure Call

1. Client procedure calls client stub in normal way
2. Client stub builds message, calls local OS
3. Client's OS sends message to remote OS
4. Remote OS gives message to server stub
5. Server stub unpacks parameters, calls server
6. Server does work, returns result to the stub
7. Server stub packs it in message, calls local OS
8. Server's OS sends message to client's OS
9. Client's OS gives message to client stub
10. Stub unpacks result, returns to client

Parameter Passing in RPC

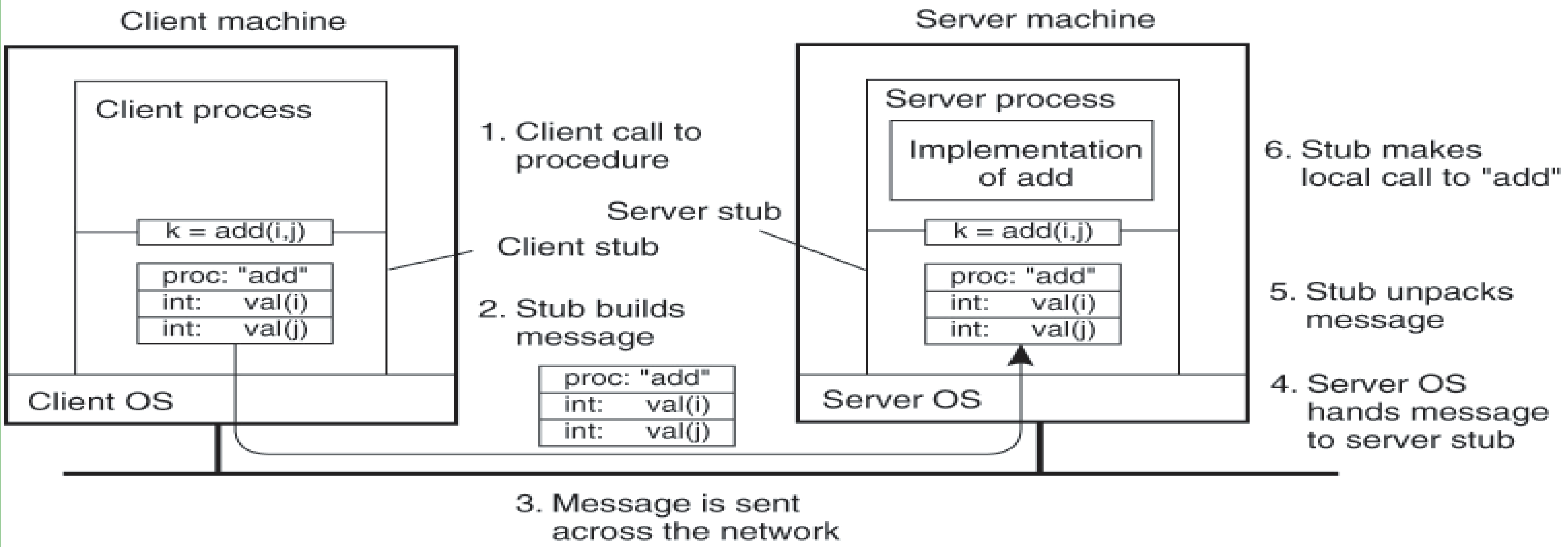
- Parameter marshalling: Packing parameters into a message
  - **Passing by value**
    - int, char
  - **Passing by reference**
    - Arrays
- IBM mainframes use EBCDIC character code, whereas IBM personal computers use ASCII
  - If machines are different, characters can be interpreted differently
- We might have similar problems in representation and interpretations of integers and floating point numbers.
- Moreover; Intel Pentium number their bytes from right to left (little endian). Whereas Sun SPARC number them the other way (big endian).

Marshalling

- **Packaging parameters is called *marshalling***
- Problem: different machines have different data formats
  - Intel: little endian, SPARC: big endian
- Solution: use a standard representation
  - Example: external data representation (XDR)

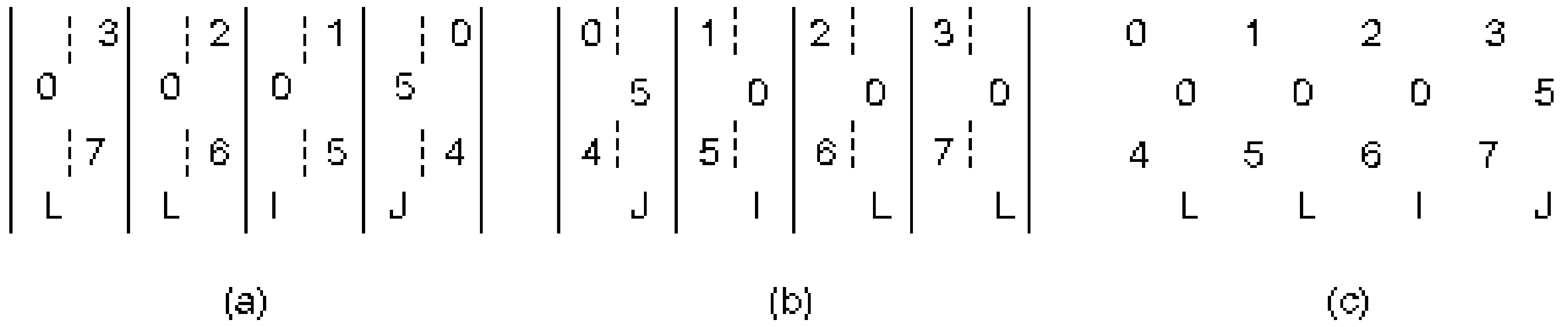
- Client stub marshals the parameters to the runtime library for transmission
- Server stub unmarshals the parameters and call the server
- The reply goes back by the reverse route

Passing Value Parameters (1)



The steps involved in a doing a remote computation through RPC

Passing Value Parameters (2)



a) Original message on x86
b) The message after receipt on the SPARC
c) The message after being inverted. The little numbers in boxes indicate the address of each byte

Passing Reference Parameters

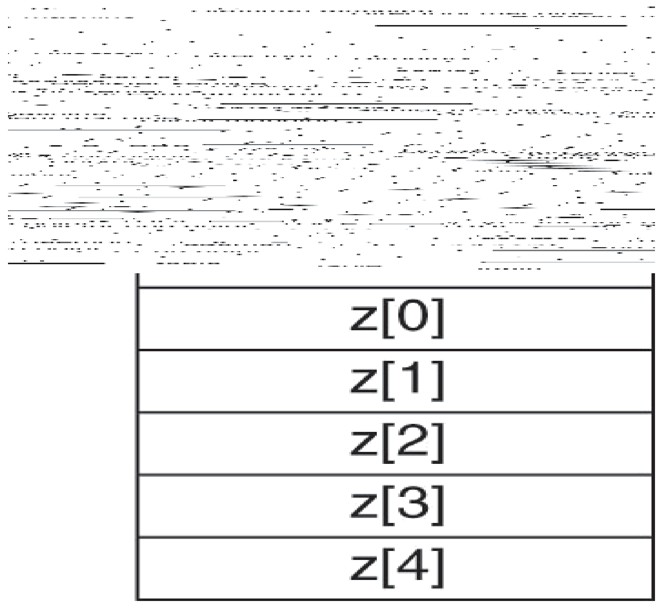
- Replace with pass by copy/restore
- Need to know size of data to copy
  - Difficult in some programming languages

- Solves the problem only partially
  - What about data structures containing pointers?
  - Access to memory in general?

Parameter Specification and Stub Generation

```
foobar( char x; float y; int z[5] )
{
    ....
}
```

(a)



(b)

(a) A procedure. (b) The corresponding message.

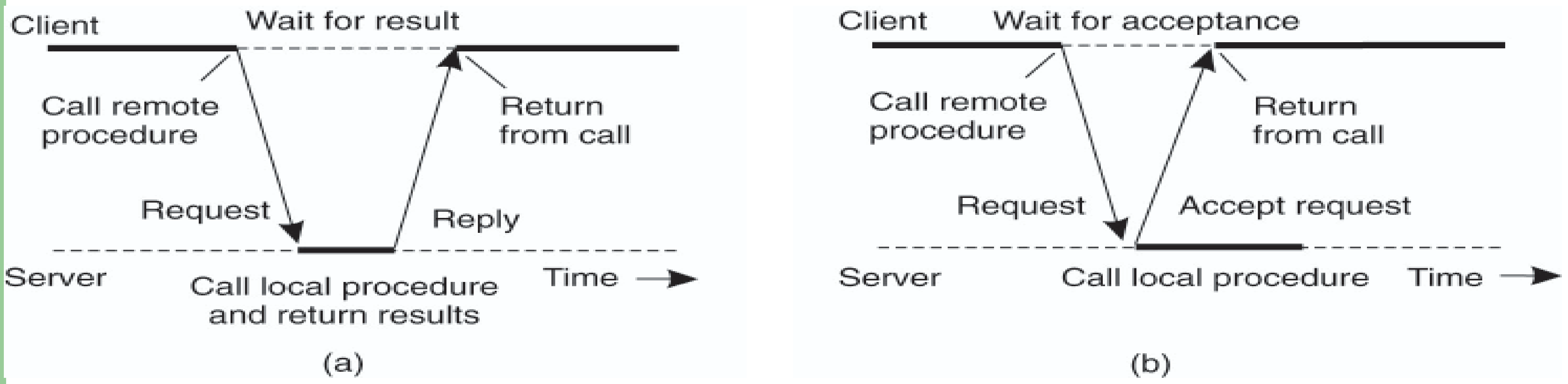
Asynchronous RPC

- RPC calls that do not block waiting for replies
- More efficient than synchronous RPC when replies are not required
- When no reply message is required
  - a client can make a RPC and proceed without waiting for the server to complete the operation
  - several client requests can be buffered and transmitted together
- When a reply message is required
  - a client can make a call, proceed with other operations and claim the reply later
- e.g., X-Window system
  - X Window manager as a server

Asynchronous RPCs

Essence: Try to get rid of the strict request-reply behavior, but let the client continue without waiting for an answer from the server.

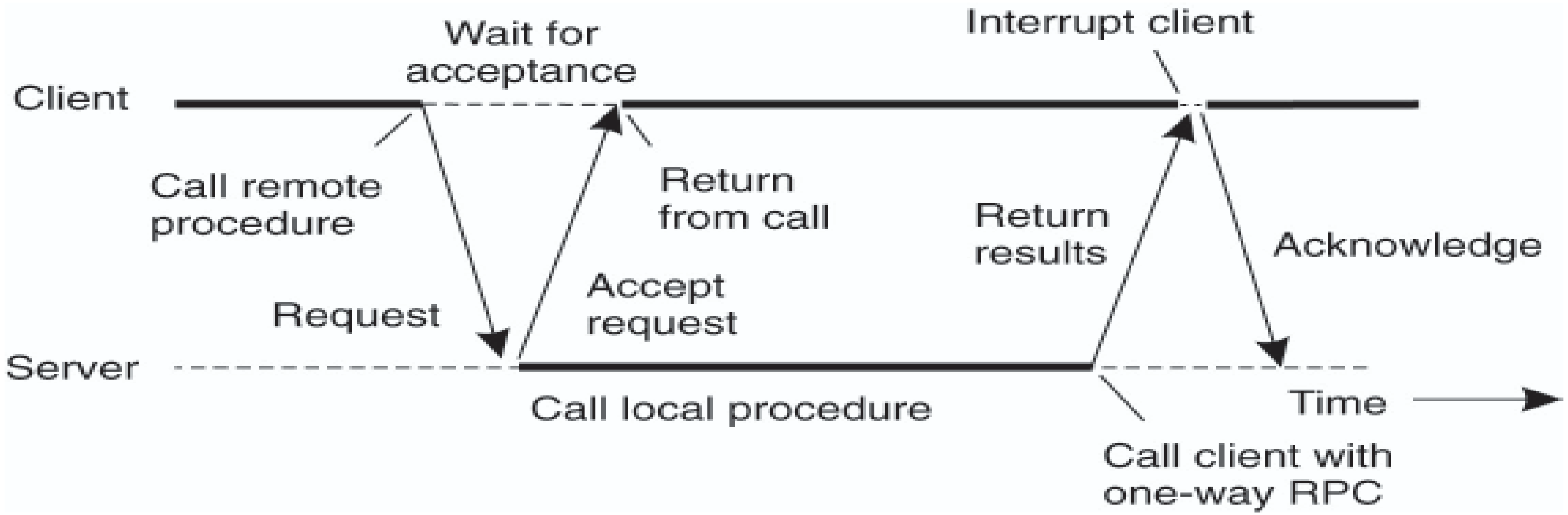
Asynchronous RPC



a) The interconnection between client and server in a traditional RPC
b) The interaction using asynchronous RPC

Deferred Synchronous RPC

A client and server interacting through two asynchronous RPCs



Case Study: SUNRPC

- One of the most widely used RPC systems
- Developed for use with NFS
- Built on top of UDP or TCP
- Multiple arguments marshaled into a single structure
- At-least-once semantics if reply received, at-least-zero semantics if no reply. With UDP tries at-most-once
- Use SUN's eXternal Data Representation (XDR)
  - Big endian order for 32 bit integers, handle arbitrarily large data structures
- XDR has been extended to become Sun RPC IDL
- An interface contains a *program number, version number, procedure definition* and *required type definitions*

SunRPC

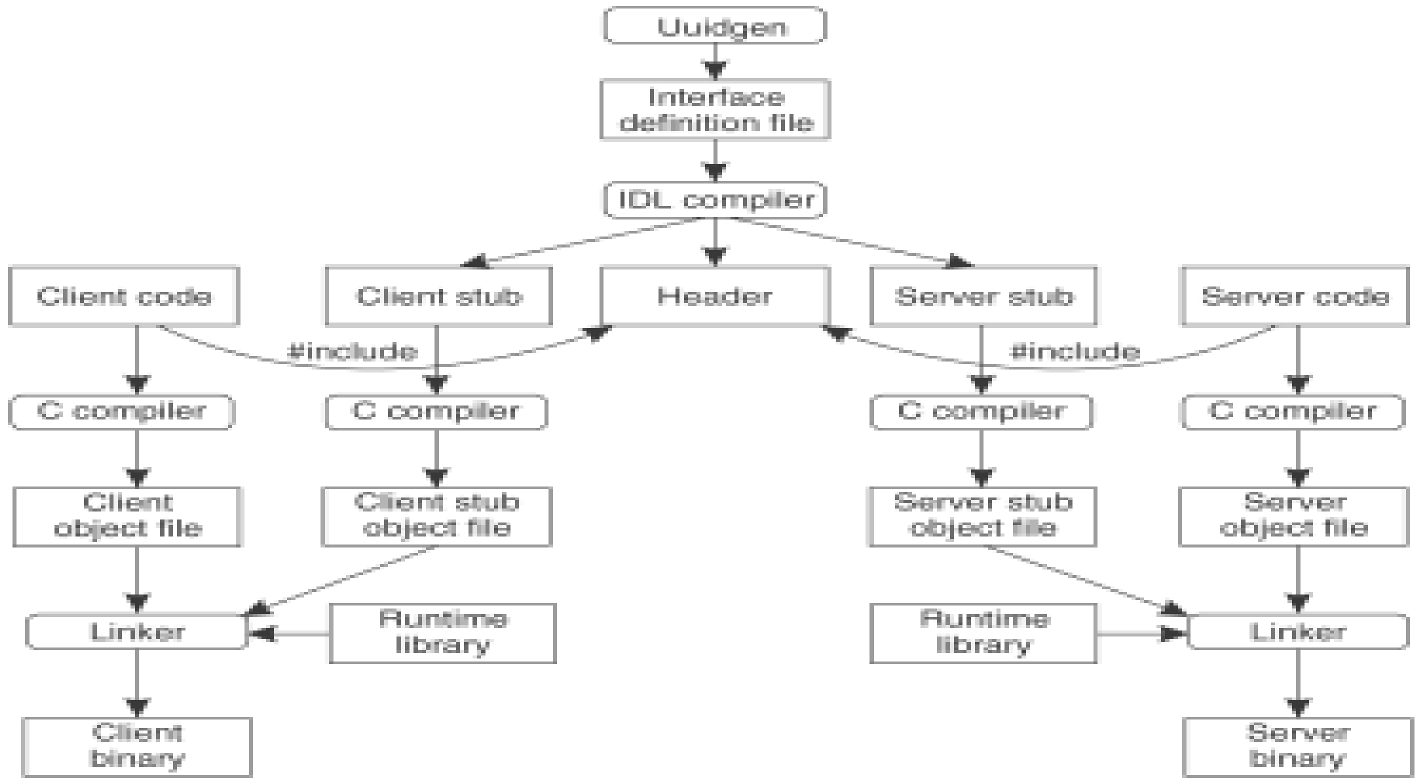
- Venerable, widely-used RPC system
- Defines “XDR” (“eXternal Data Representation”) -- C-like language for describing functions -- and provides a compiler that creates stubs

```
struct fooargs {
  string msg<255>;
  int baz;
}
```

Case Study: DCE/RPC

- Distributed Computing Environment / Remote Procedure Calls
- DCE/RPC was commissioned by the Open Software Foundation
- Client-server – runtime semantics
  - – Run at most once
  - – Defining remote procedure as idempotent

Implementation Model for DCE

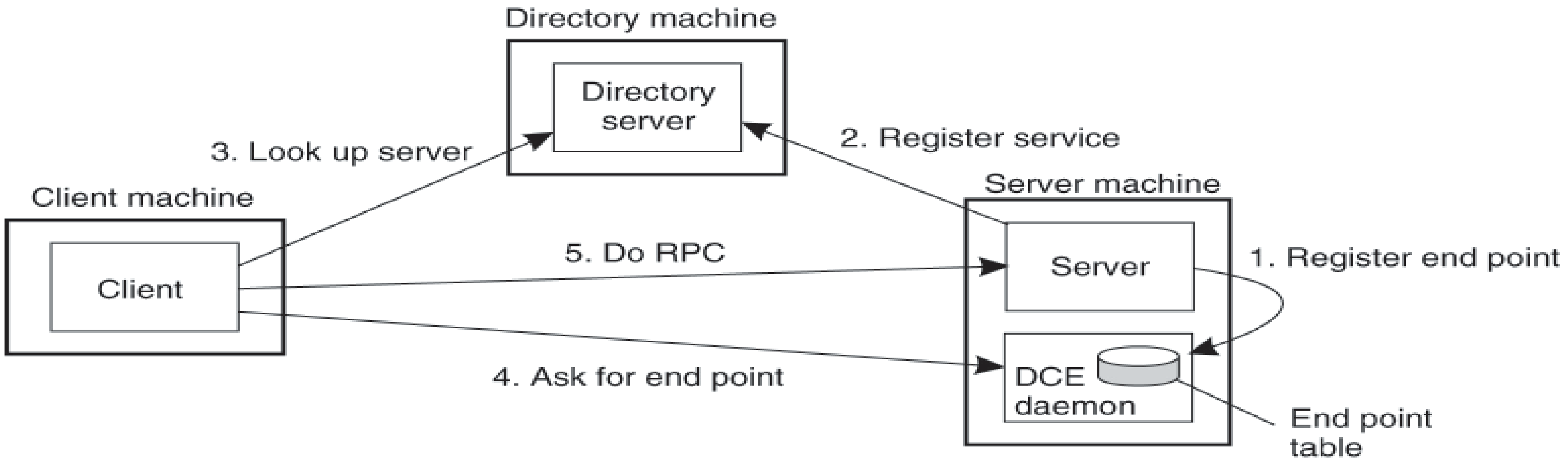


Binding a Client to a Server

- Binding is the process of connecting the client to the server
  - the server, when it starts up, exports its interface
    - identifies itself to a *network name server*
    - tells *RPC runtime* that it is alive and ready to accept calls
  - the client, before issuing any calls, imports the server
    - RPC runtime uses the name server to find the location of the server and establish a connection
- The import and export operations are explicit in the server and client programs

Binding a Client to a Server

- Registration of a server makes it possible for a client to locate the server and bind to it
- Server location is done in two steps:
  - Locate the server's machine.
  - Locate the server on that machine.



Conclusion

RPC provides programmers with a familiar mechanism for building distributed systems. RPC facility is not an universal panacea for all types of distributed applications, it does provide a valuable communication mechanism that is suitable for building a fairly large number of distributed applications.