

I226

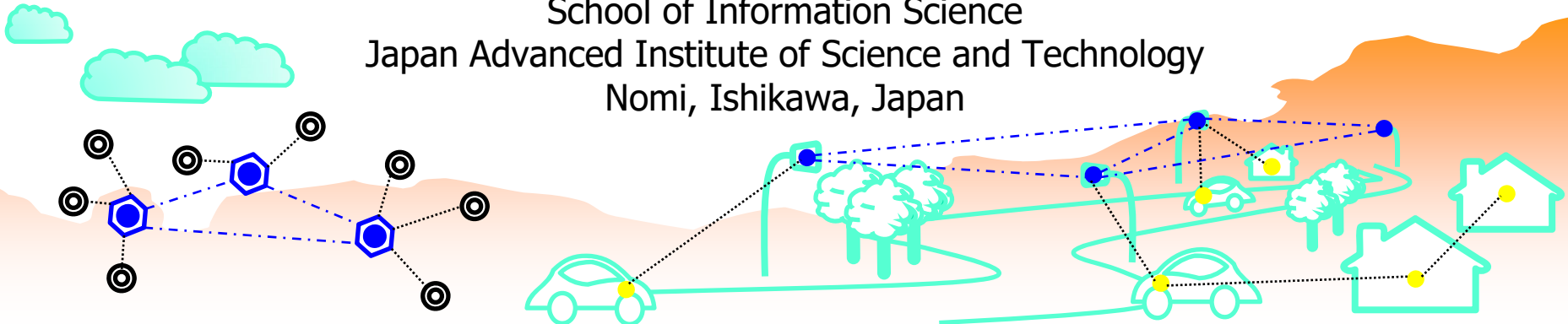
Computer Networks

Chapter 7

Application Layer and Network Programming Basics

Assoc. Prof. Yuto Lim

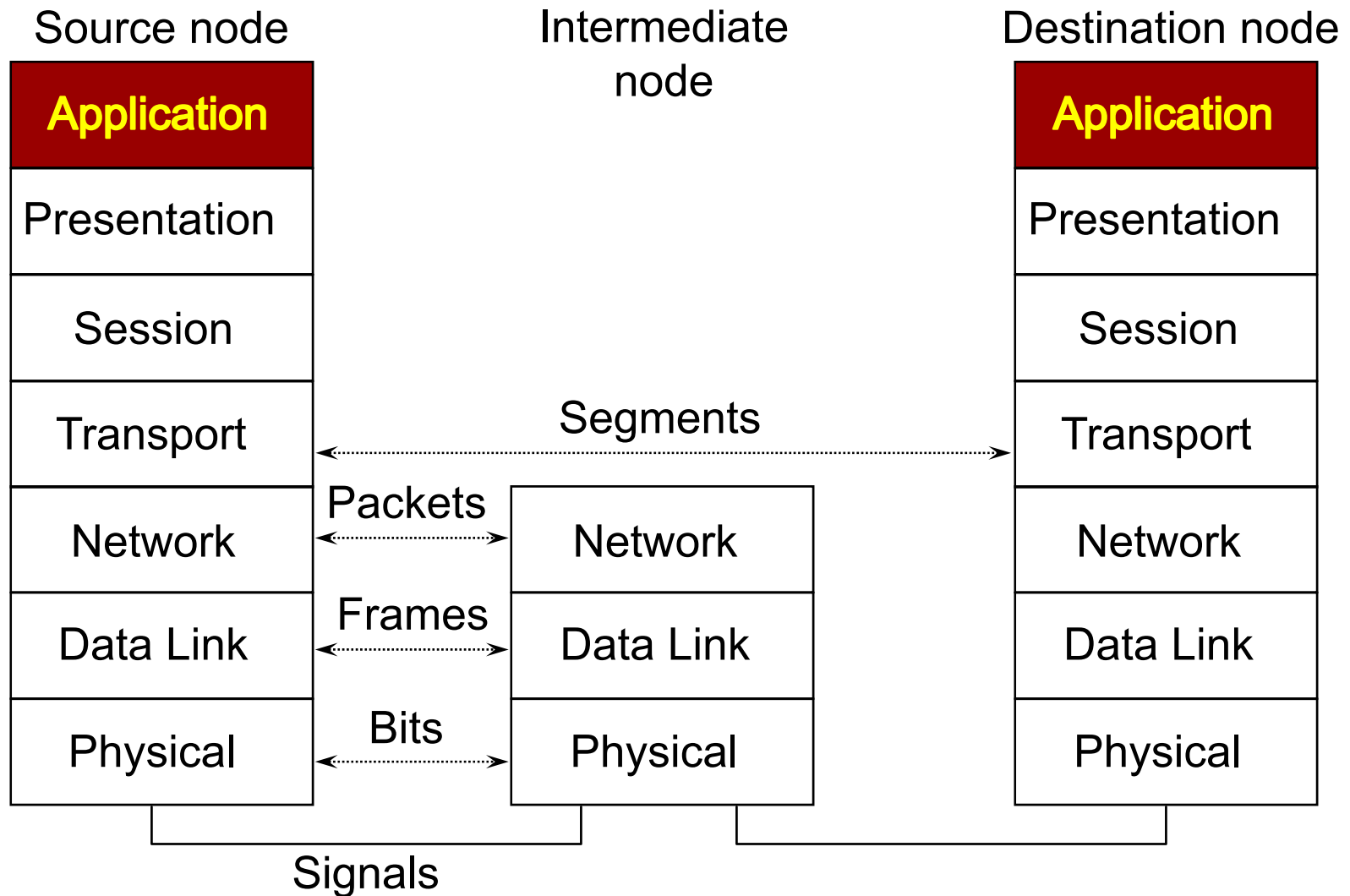
School of Information Science
Japan Advanced Institute of Science and Technology
Nomi, Ishikawa, Japan



Objectives of this Chapter

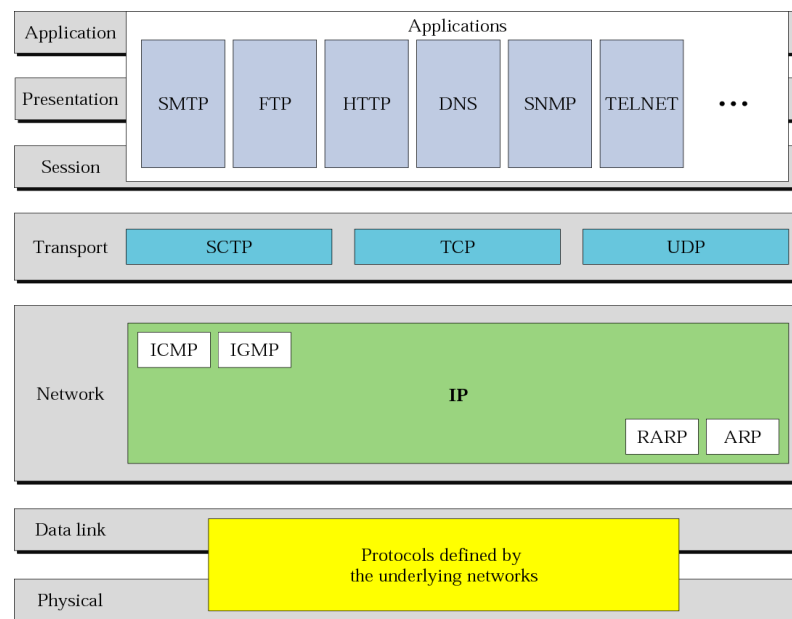
- Give an understanding what is the application layer and its three architectures
- Provide the knowledge of application layer protocols: SMTP, FTP, HTTP, DNS, SNMP and TELNET
- Explain the socket programming and demonstrate how the socket programming work
- Also explain the remote procedure call

Application Layer



Outline

- Introduction to Network Applications
- Electronic Mail
 - SMTP, POP3, IMAP
- FTP
- Web and HTTP
- DNS
- SNMP
- TELNET
- Socket Programming
- Remote Procedure Call



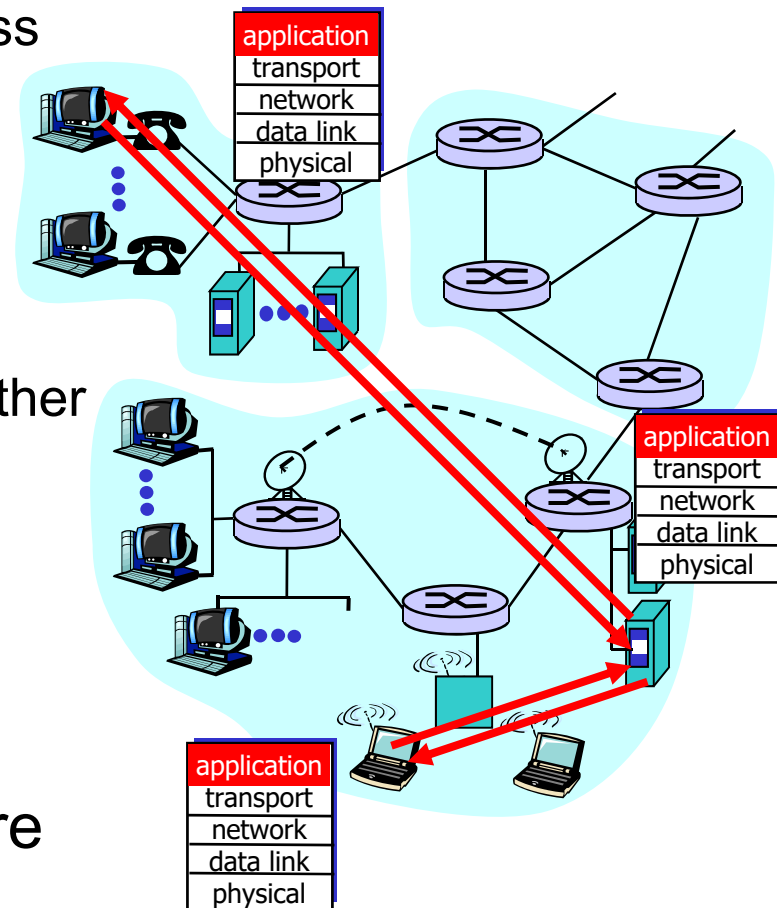
* SCTP = Stream Control Transport Protocol

Introduction to Application Layer

- *Note:* Although the ISO OSI Reference Model defines a session and presentation layer, they are often integrated into some other layer in practice. In many cases they are simply not used
- Duties of network layer
 - **Transfer** bits/bytes
 - Operates at the application's request
- Duties of application layer
 - **Meaning** of bits/bytes
 - What data to transmit
 - When to transmit data
 - Where to transmit data to
- Application architectures:
 - Client-server
 - Peer-to-peer (P2P)
 - Hybrid of client-server and P2P

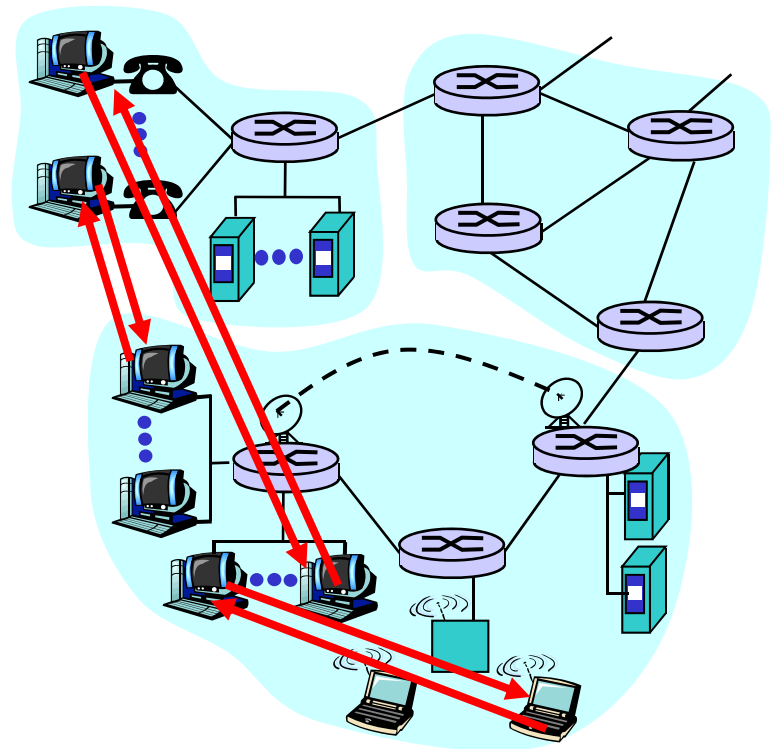
Client-Server Architecture

- **Server**
 - ▣ Always-on host, **permanent** IP address
 - ▣ Server grows in scale (massive data center)
- **Client**
 - ▣ Intermittently connected with server
 - ▣ **Dynamic** IP addresses
 - ▣ No communicate directly with each other
- **Example:**
 - ▣ Social networking, e.g., **Facebook** (4 February 2004)
 - ▣ Video sharing, e.g., **YouTube** (14 February 2005)
- **Pros:** Highly accessible and secure
- **Cons:** Congested



P2P Architecture

- No always on server
- Arbitrary end systems (**peers**) directly communicate
 - ▣ Peers are intermittently connected and change IP addresses
- Examples:
 - ▣ File sharing, e.g., **Gnutella** (14 March 2000)
 - ▣ Video conferencing, e.g., **Skype** (29 August 2003)
 - ▣ Instant messaging, e.g., **LINE** (March 2011)
- Pros: Highly scalable
- Cons: Difficult to manage



Hybrid of Client-Server and P2P

File sharing: **Napster, DropBox**

- File transferring is P2P, where the entire network relied on the central server
- File search centralized
 - Peers register content at central server
 - Peers query same central server to locate content

Instant messaging: **Windows Live Messenger, Twitter**

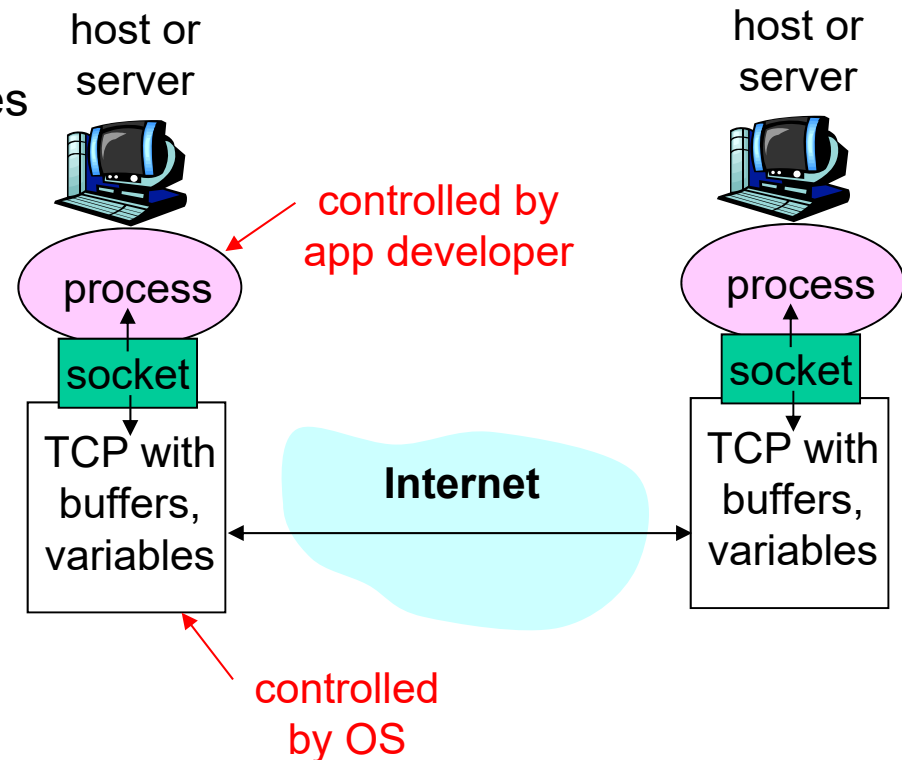
- Chatting between two users is P2P
- Presence detection centralized
 - User registers its IP address with central server when it comes online
 - User contacts central server to find IP addresses of associates

Processes and Sockets

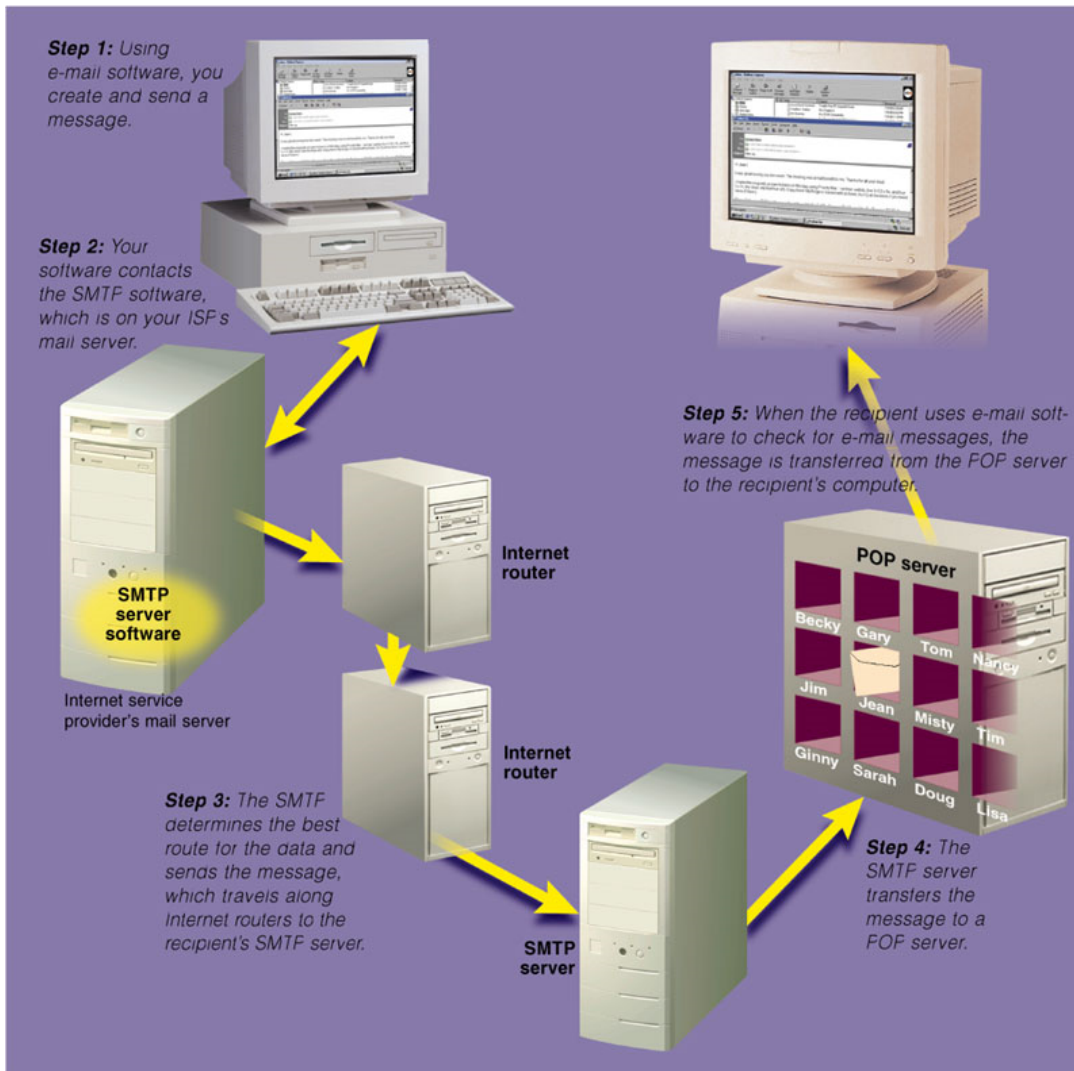
- **Process**: program running within a host
- Within same host, 2 processes communicate using inter-process communication (defined by OS)
- Processes in different hosts communicate by exchanging messages
- Process sends/receives messages to/from its **socket**, which analogous to door
 - ▢ Sending process shoves message out door
 - ▢ Sending process relies on transport infrastructure on other side of door which brings message to socket at receiving process
- API:
 - ▢ Choice of transport protocol
 - ▢ Ability to fix a few parameters

Client process: process that initiates communication

Server process: process that waits to be contacted



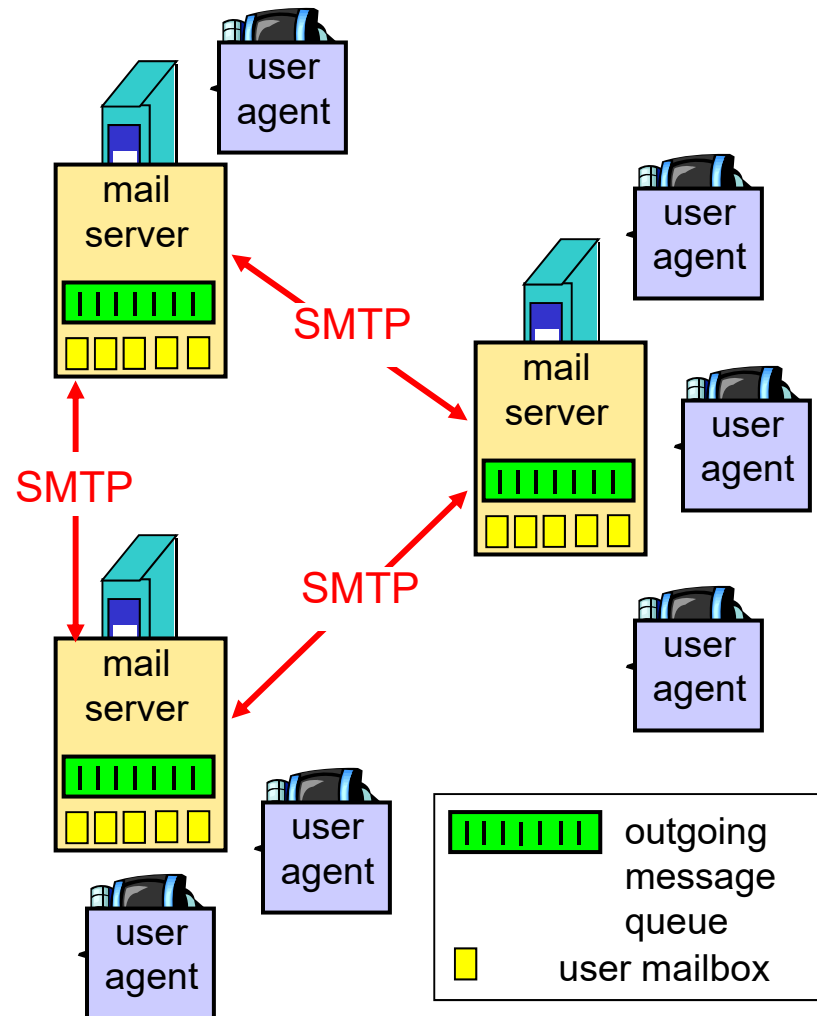
How Does Email Works?



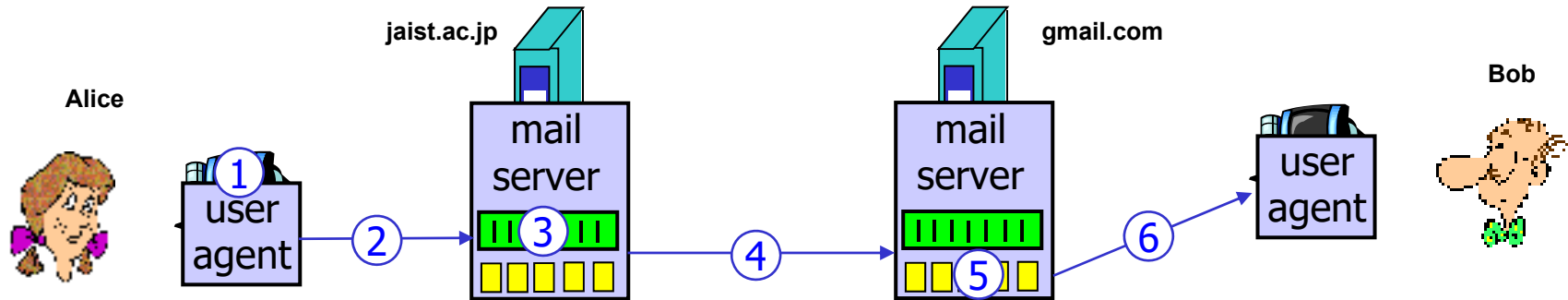
- E-mails are transferred across the Internet via **SMTP**
- Mail server uses SMTP to determine how to route the message through the Internet and then sends the message
- When the message arrives at the recipient's mail server, the message is transferred to a **POP3 server**. (POP = Post Office Protocol)
- POP server holds the message until the recipient retrieves it with his/her email software

Electronic Mail

- 3 components
 - User agents
 - Mail servers
 - Simple mail transfer protocol (SMTP)
- User agent
 - Also known as (a.k.a.) “mail reader”
 - Composing, editing, reading mail messages
 - E.g., Eudora, MS Outlook
 - Outgoing, incoming messages stored on server
- Mail servers
 - Mailbox contains incoming messages for user
 - Message queue of outgoing (to be sent) mail messages
- SMTP
 - A protocol between mail servers to send email messages
 - Client: sending mail server
 - Server: receiving mail server



Simple Mail Transfer Protocol (SMTP)



- Uses TCP to reliably transfer email message from client to server, **port 25**
- Direct transfer: sending server to receiving server, 3 phases of transfer
 - Handshaking (greeting), transfer of messages, closure
- Command/response interaction
 - Commands: ASCII text
 - Response: status code and phrase
- Messages must be in 7-bit ASCII
- SMTP: Request for Comments (RFC) 2821

- ① Alice uses MS Outlook to compose message "to" bob@gmail.com
- ② Alice's MS Outlook sends message to her mail server; message placed in message queue
- ③ Client side of SMTP opens TCP connection with Bob's mail server
- ④ SMTP client sends Alice's message over the TCP connection
- ⑤ Bob's mail server places the message in Bob's mailbox
- ⑥ Bob invokes his user agent to read message

```
S: 220 gmail.com opening connection
C: HELLO jaist.ac.jp
S: 250 Hello jaist.ac.jp, pleased to meet you
C: MAIL FROM: <alice@jaist.ac.jp>
S: 250 alice@jaist.ac.jp ... Sender ok
C: RCPT TO: <bob@gmail.com>
S: 250 bob@gmail.com ... Recipient ok
C: DATA
S: 354 Enter mail, end with "." on a line by itself
C: Do you like ketchup?
C: How about pickles?
C: ...
S: 250 Message accepted for delivery
C: QUIT
S: 221 gmail.com closing connection
```

Figure: SMTP interaction

POP3 Protocol

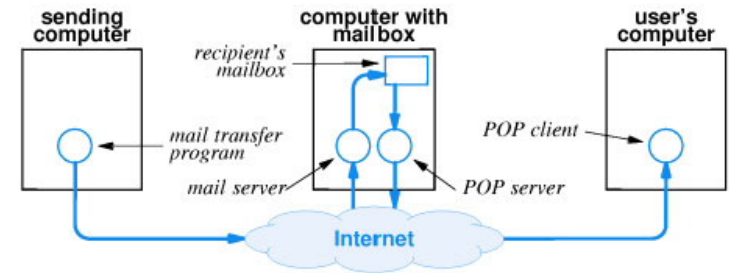
- Current version is POP3 (POP Version 3), which is stateless across sessions

Authorization Phase:

- Client commands
 - user: declare username
 - pass: password

Transaction Phase:

- Client commands
 - list: list message numbers
 - retr: retrieve message by number
 - dele: delete
 - quit
- In both phase, server responses
 - +OK
 - ERR
- Previous example uses “download-and-delete” mode. Bob cannot re-read email if he changes client. “download-and-keep”: copies of messages on different clients

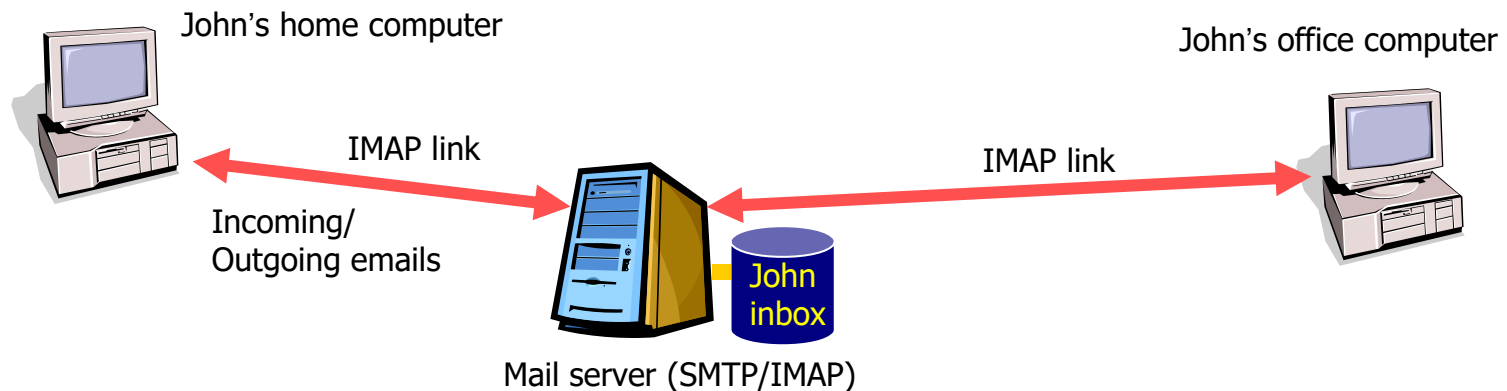


```
S: +OK POP3 server ready
C: user bob
S: +OK
C: pass hungry
S: +OK user successfully logged on

C: list
S: 1 498
S: 2 912
S: .
C: retr 1
S: <message 1 contents>
S: .
C: dele 1
C: retr 2
S: <message 1 contents>
S: .
C: dele 2
C: quit
S: +OK POP3 server signing off
```

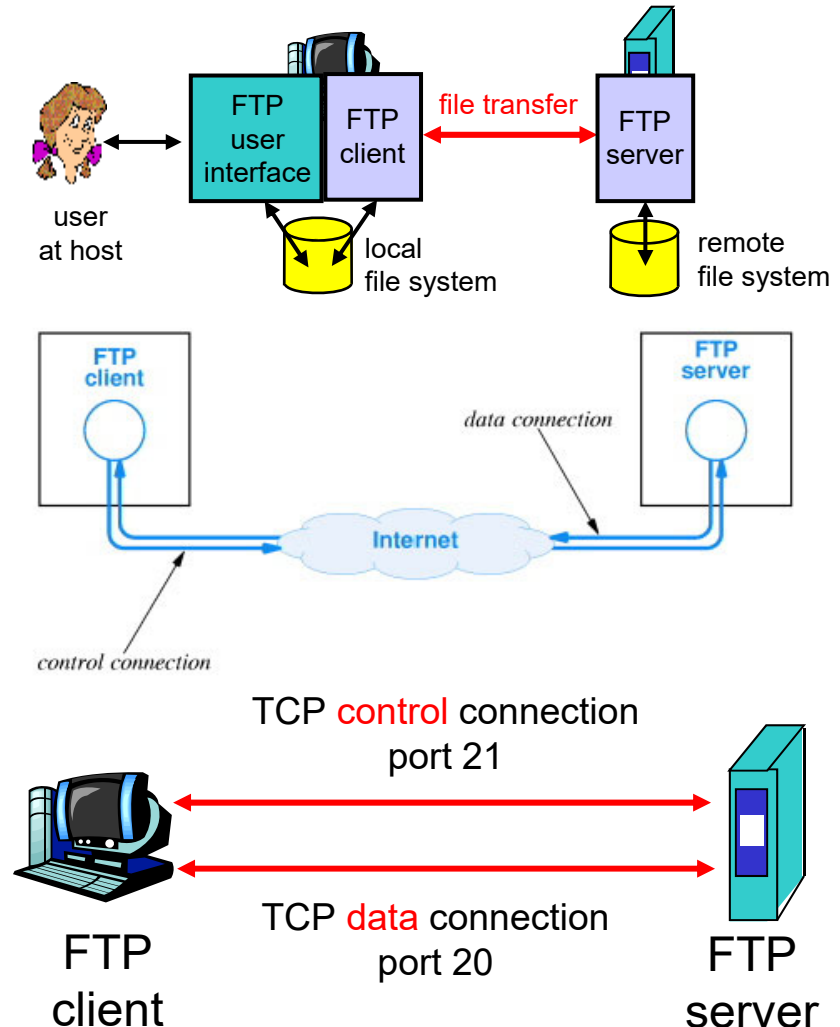
Interactive Mail Access Protocol (IMAP)

- Another popular method by which users obtain their emails is called **central mail spool**, **IMAP** (e.g., **JAIST Webmail** Service using Zimbra)
- Keep all messages in one place: at the server
- Allows user to organize messages in folders
- IMAP keeps user state across sessions
 - Names of folders and mappings between message IDs and folder name
- IMAP **encrypts** passwords so that someone *sniffing* the network cannot directly obtain his password



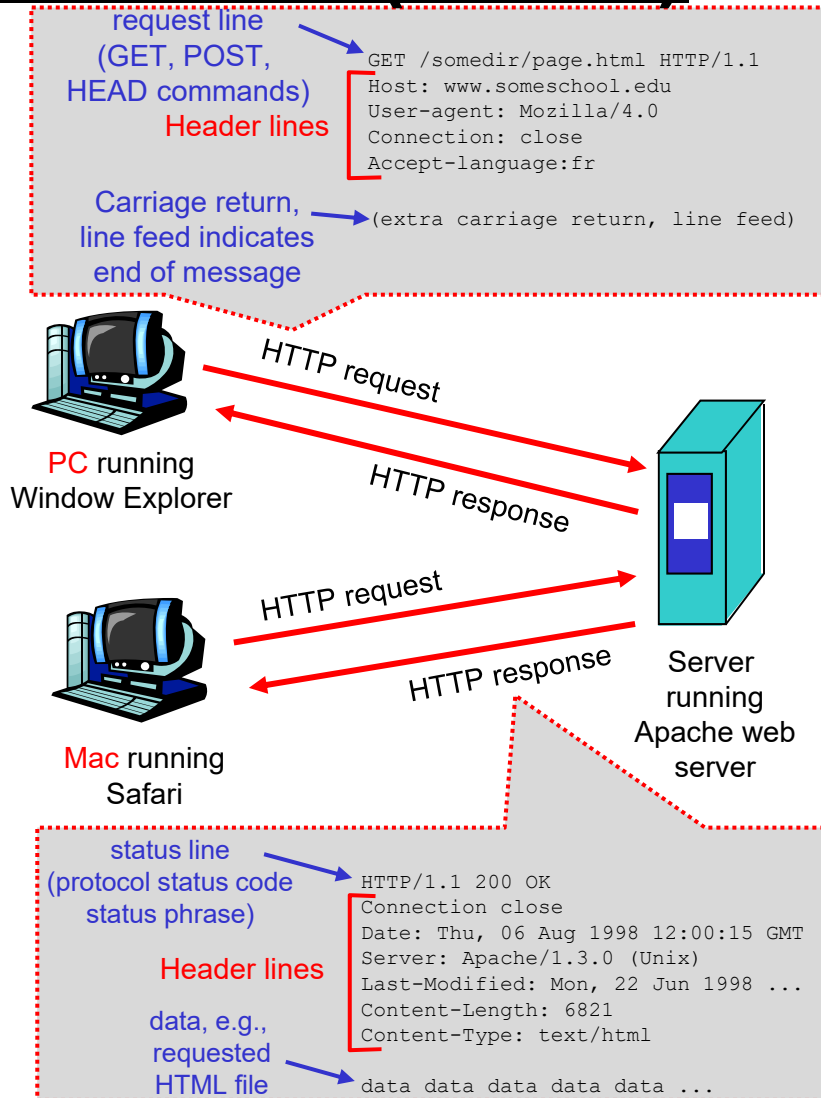
File Transfer Protocol (FTP)

- Transfer file to/from remote host
- Client/server model
 - Client: side that initiates transfer (either to/from remote)
 - Server: remote host
- FTP operation:
 - Client initiates TCP control connection to FTP server, **port 21**
 - Client obtains authorization over control connection and client browses remote directory by sending commands over control connection
 - When server receives a command for a file transfer, the server opens a TCP data connection to client, **port 20**
 - After transferring one file, server closes connection. Server opens a second TCP data connection to transfer another file
 - FTP server maintains “state”: current directory, earlier authentication
- FTP: RFC 959



Hypertext Transfer Protocol (HTTP)

- Web's application layer protocol
- Client/server model
 - Client: browser that requests, receives, "displays" Web objects
 - Server: web server sends objects in response to requests
- HTTP operation
 - Client initiates TCP connection (creates socket) to server, **port 80**
 - Server accepts TCP connection from client
 - HTTP messages (application-layer protocol messages) exchanged between browser (HTTP client) and Web server (HTTP server)
 - TCP connection closed
- HTTP is "stateless"
 - Server maintains no information about past client requests
- HTTP 1.0: RFC 1945, HTTP 1.1: RFC 2068



Domain Name System (DNS)

- People have many identifiers: name, passport number
- Internet uses hosts, routers
 - IP address (32 bit) – used for addressing datagrams
 - “name”, e.g., www.yahoo.com – used by humans
- How to map between IP addresses and name?
 - Answer: **DNS**
 - Distributed database implemented in hierarchy of many name servers
 - Application-layer protocol host, routers, name servers communicate to resolve names (address/name translation)
 - Note: core Internet function, implemented as application-layer protocol
 - Complexity at network’s “edge”
- DNS is further discussed on **Chapter 8**

SNMP

- **Simple network management protocol** (SNMP) provides facilities for managing and monitoring network resources on the Internet
- **SNMP components**
 - SNMP agents (**port 161**) is software that runs on a piece of network equipment (host, router, printer, or others) and that maintains information about its configuration and current state in a database
 - SNMP managers (**port 162**) is an application program that contacts an SNMP agent to query or modify the database at the agent
 - **Management information bases** (MIBs) describes the information in the database
 - SNMP protocol is the application layer protocol used by SNMP agents and managers to send and receive data

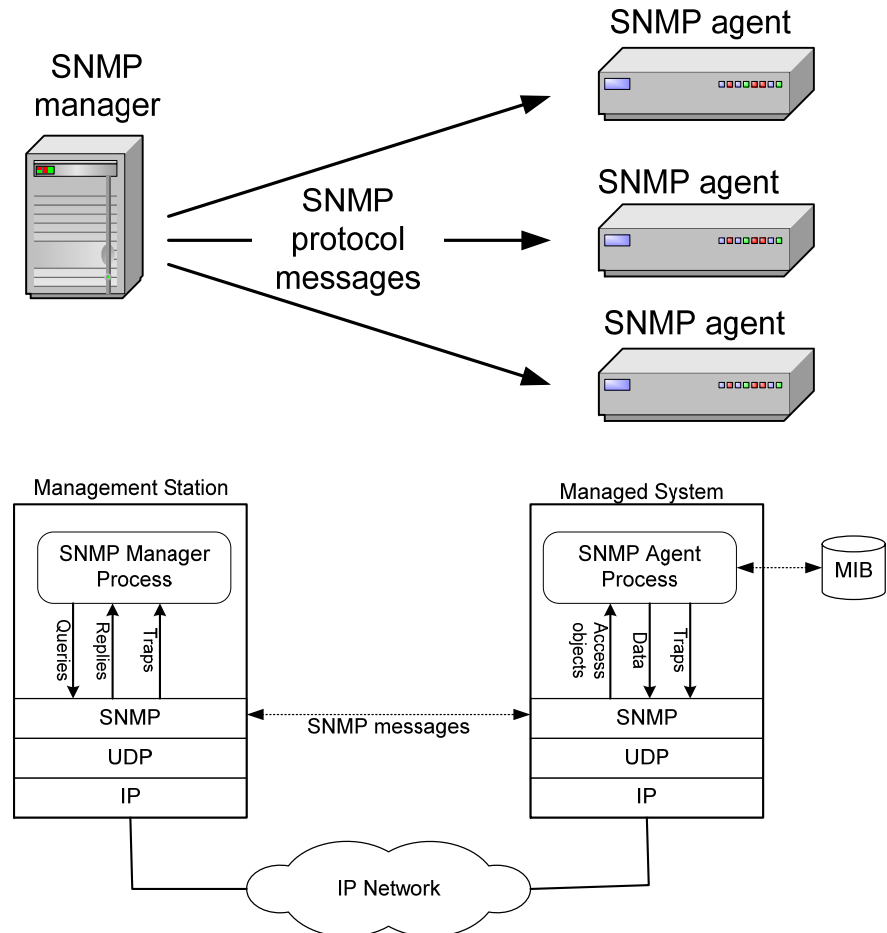
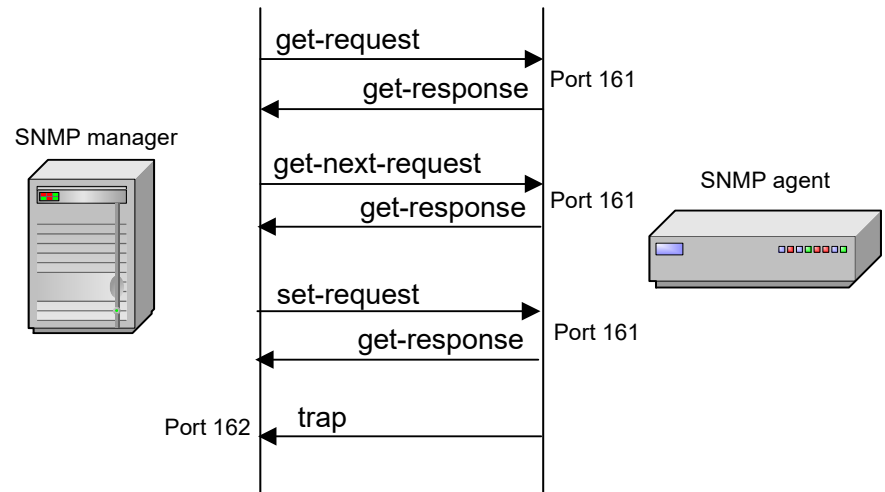


Figure: SNMP Interactions

SNMP Protocol

- SNMP manager and an SNMP agent communicate using the SNMP protocol
 - General: Manager sends queries and agent responds
 - Exception: **Traps** are initiated by agent
- Traps are messages that are asynchronously sent by an agent to a manager
- Traps are triggered by an event
- Defined traps include
 - **linkDown**: event that an interface went down
 - **coldStart**: unexpected restart (i.e., system crash)
 - **warmStart**: soft reboot
 - **linkUp**: the opposite of linkDown
 - **AuthenticationFailure**



get-request: requests the values of one or more objects

get-next-request: requests the value of the next object, according to a lexicographical ordering of OIDs

set-request: a request to modify the value of one or more objects

get-response: sent by SNMP agent in response to a *get-request*, *get-next-request*, or *set-request* message

trap: an SNMP trap is a notification sent by an SNMP agent to an SNMP manager, which is triggered by certain events at the agent

TELNET: Remote Login

- TELNET is a protocol that provides a general, bi-directional, 8-bit byte oriented communications facility
- `telnet` is a program that supports the TELNET protocol over TCP
- Many application protocols are built upon the TELNET protocol
- TELNET: RFC 854
- Data and control over the same connection
- **Network Virtual Terminal (NVT)**
 - ▣ Intermediate representation of a generic terminal
 - ▣ Provides a standard language for communication of terminal control functions

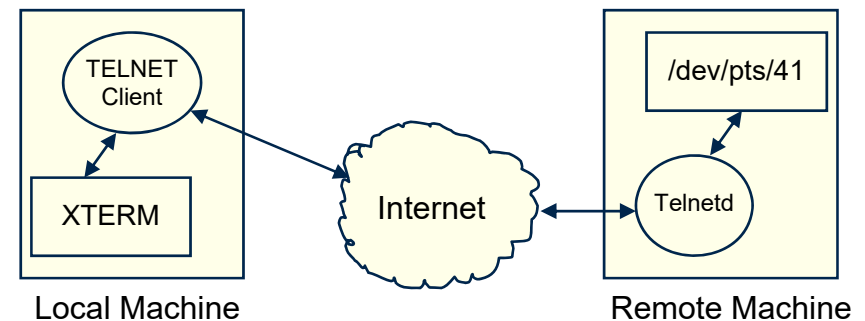
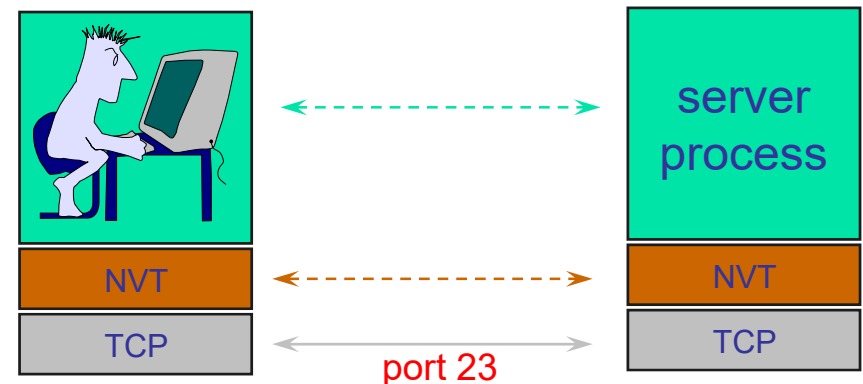


Figure: The telnet server copies data to/from the connection data to a pseudo tty in the remote machine using a bidirectional channel



Network Programming

- The way network applications are developed has evolved
- The evolution of network programming is parallel to the evolution of computer languages

Network API	Equivalent Language
Sockets	Assembly Language
RPC (Remote Procedure Call)	Procedural Language (C, Pascal, Fortran)
RMI (Remote Method Invocation) also called Remote Objects	Object Oriented Languages (C++, Java, C#)

Socket Programming

Socket

a *host-local, application-created, OS-controlled* interface (a “door”) into which application process can **both send and receive** messages to/from another application process

Socket API:

- Introduced in BSD4.1 UNIX, 1981
- Explicitly created, used, released by apps
- Client/server paradigm
- 2 types of transport service via socket API
 - UDP, unreliable datagram
 - TCP, reliable byte stream-oriented

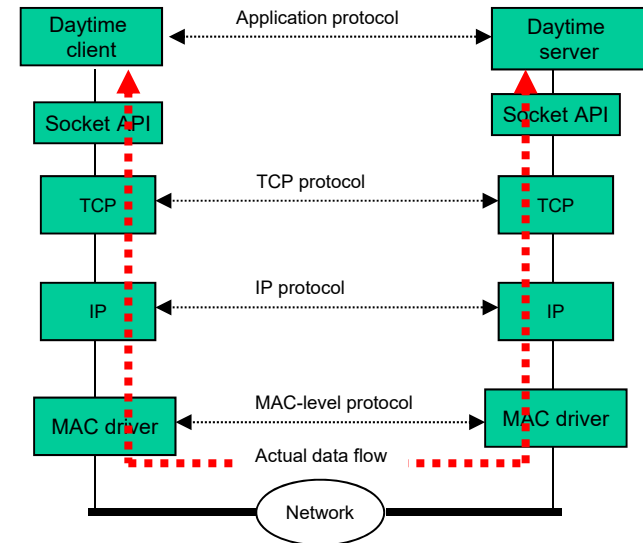


Figure: Socket API Location in the Protocol Stack

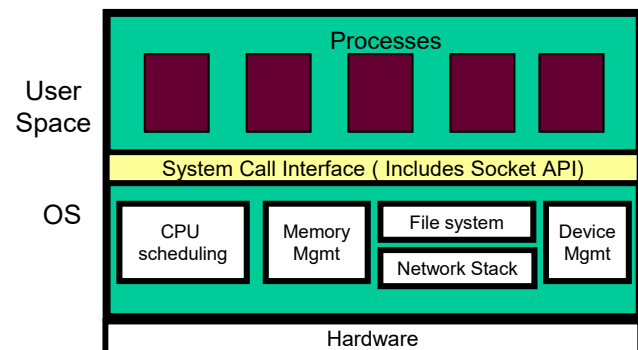
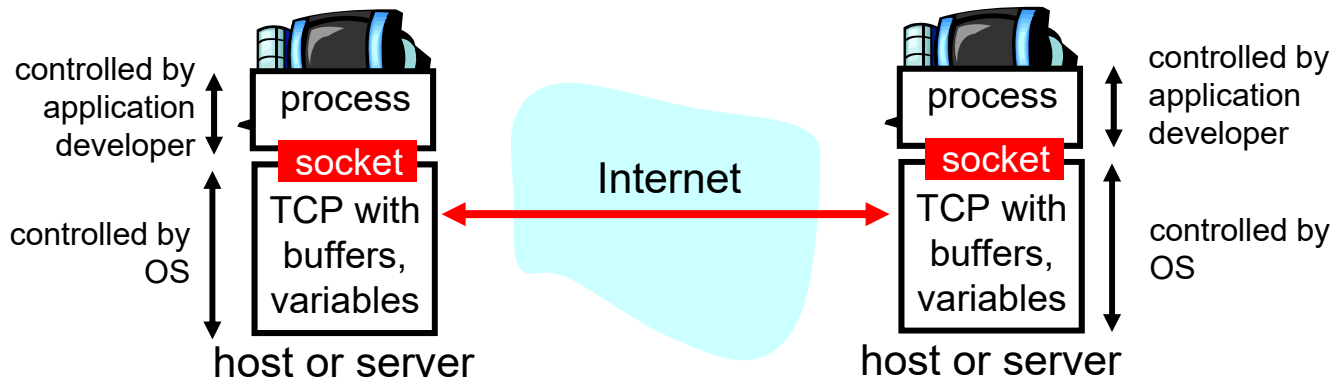


Figure: Socket API Location in the OS

Socket Programming using TCP



- TCP service: reliable transfer of bytes from one process to another
- IP addresses: 4 decimal values in the range 0~255, e.g.,: 140.179.220.200
- Ports: 16-bit to identify the application process that is a network endpoint
 - ▣ **Reserved ports** or **well-known ports** (0 to 1023), e.g.,: FTP(21), TELNET (23), etc
 - ▣ **Ephemeral ports** (1024-65535) for ordinary user-developed programs
- Associations:
 - **Half-association** (or socket address) is the **triple (3-tuple)**
 {protocol, local-IP, local-port}, e.g.,: {tcp, 130.245.1.44, 23}
 - **Full-association** is the **5-tuple** that completely specifies the 2 end-points of a connection
 {protocol, local-IP, local-port, remote-IP, remote-port}, e.g.:
 {tcp, 130.245.1.44, 23, 130.245.1.45, 1024}

Socket Programming with TCP

- Client must contact server
 - ▣ Server process must first be running
 - ▣ Server must have created socket (door) that welcomes client's contact

- Client contacts server by
 - ▣ Creating client-local TCP socket
 - ▣ Specifying IP address, port number of server process
 - ▣ When client creates socket: client TCP establishes connection to server TCP

- When contacted by client, server TCP creates new socket for server process to communicate with client
 - ▣ Allows server to talk with multiple clients
 - ▣ Source port numbers used to distinguish clients

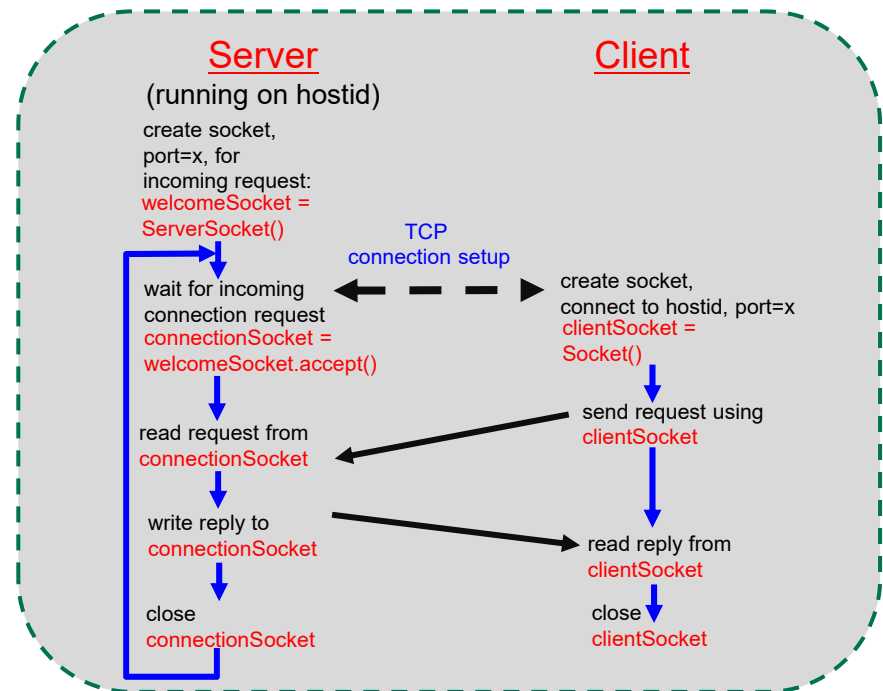
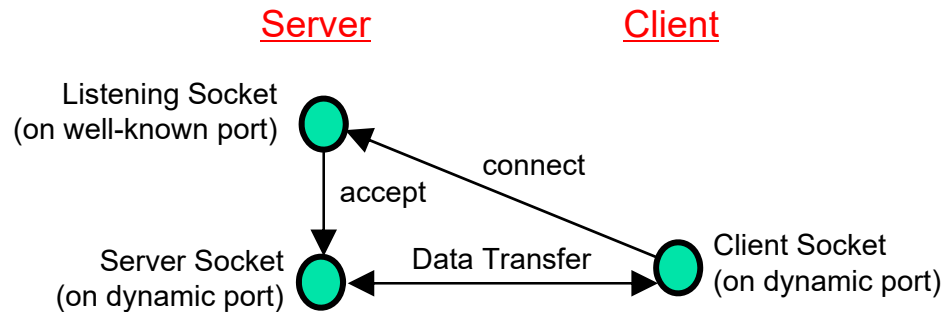
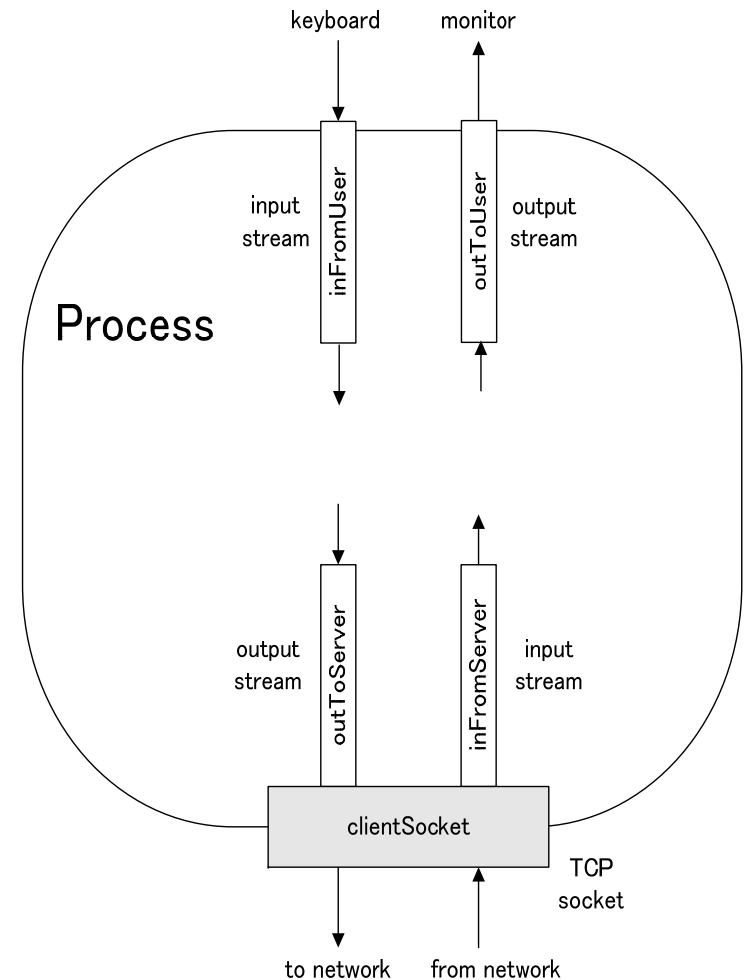


Figure: Client-Server Interaction

Socket Programming with TCP (cont.)

- Definition:
 - **Stream** is a sequence of characters that flow into or out of a process
 - Input stream is attached to some input source for the process, e.g., keyboard or socket
 - Output stream is attached to an output source, e.g., monitor or socket
- Example client-server application:
 - ① Client reads line from standard input (**inFromUser** stream), sends to server via socket (**outToServer** stream)
 - ② Server reads line from socket
 - ③ Server converts line to uppercase, sends back to client
 - ④ Client reads and prints modified line from socket (**inFromServer** stream), sends to monitor (**outToUser** stream)



JAVA Example: Server, TCP

```
import java.net.*;
import java.io.*;
class Server {
    public static void main(String[] args) throws Exception {
        <Summary Note>
        The steps to get a server up and running are shown below:
        (1) Create a server socket
        (2) Name the socket
        (3) Prepare the socket to listen
        (4) Wait for a request to connect, a new client socket is created here
        (5) Read data sent from client
        (6) Send data back to client
        (7) Close client socket
        (8) Loop back if not told to exit
        (9) Close server socket is exit command given by client
    }
}
```

```
ServerSocket welcomeSocket = new ServerSocket(6789);

while(true) {
    Socket connectionSocket = welcomeSocket.accept();

    BufferedReader inFromClient =
        new BufferedReader(new
            InputStreamReader(connectionSocket.getInputStream()));

    DataOutputStream outToClient =
        new DataOutputStream(connectionSocket.getOutputStream());

    clientSentence = inFromClient.readLine();

    capitalizedSentence = clientSentence.toUpperCase() + '\n';

    outToClient.writeBytes(capitalizedSentence);
}
}
```

Create welcoming socket at port 6789

Wait, on welcoming socket for contact by client

Create input stream, attached to socket

Create output stream, attached to socket

Read in line from socket

Write out line to socket

End of while loop, loop back and wait for another client connection

JAVA Example: Client, TCP

```

import java.io.*;
import java.net.*;

class Client {
    public static void main(String[] args) throws Exception {
        // Create input stream
        BufferedReader inFromUser =
            new BufferedReader(new InputStreamReader(System.in));

        // Create client socket, connect to server
        Socket clientSocket = new Socket("hostname", 6789);

        // Create output stream attached to socket
        DataOutputStream outToServer =
            new DataOutputStream(clientSocket.getOutputStream());

        // Create input stream attached to socket
        BufferedReader inFromServer =
            new BufferedReader(new
                InputStreamReader(clientSocket.getInputStream()));

        // Send line to server
        String sentence = inFromUser.readLine();
        outToServer.writeBytes(sentence + '\n');

        // Read line from server
        String modifiedSentence = inFromServer.readLine();

        System.out.println("FROM SERVER: " + modifiedSentence);

        clientSocket.close();
    }
}

```

<Summary Note>
The steps to get a client communicating with the server are shown below:

- (1) Create a socket with the server IP address
- (2) Connect to the server, this step also names the socket
- (3) Send data to the server
- (4) Read data returned (echoed) back from the server
- (5) Close the socket

Socket programming with UDP

- UDP: no “connection” between client and server
 - ▣ No handshaking
 - ▣ Sender explicitly attaches IP address and port of destination to each packet
 - ▣ Server must extract IP address, port of sender from received packet
- UDP: transmitted data may be received out of order, or lost

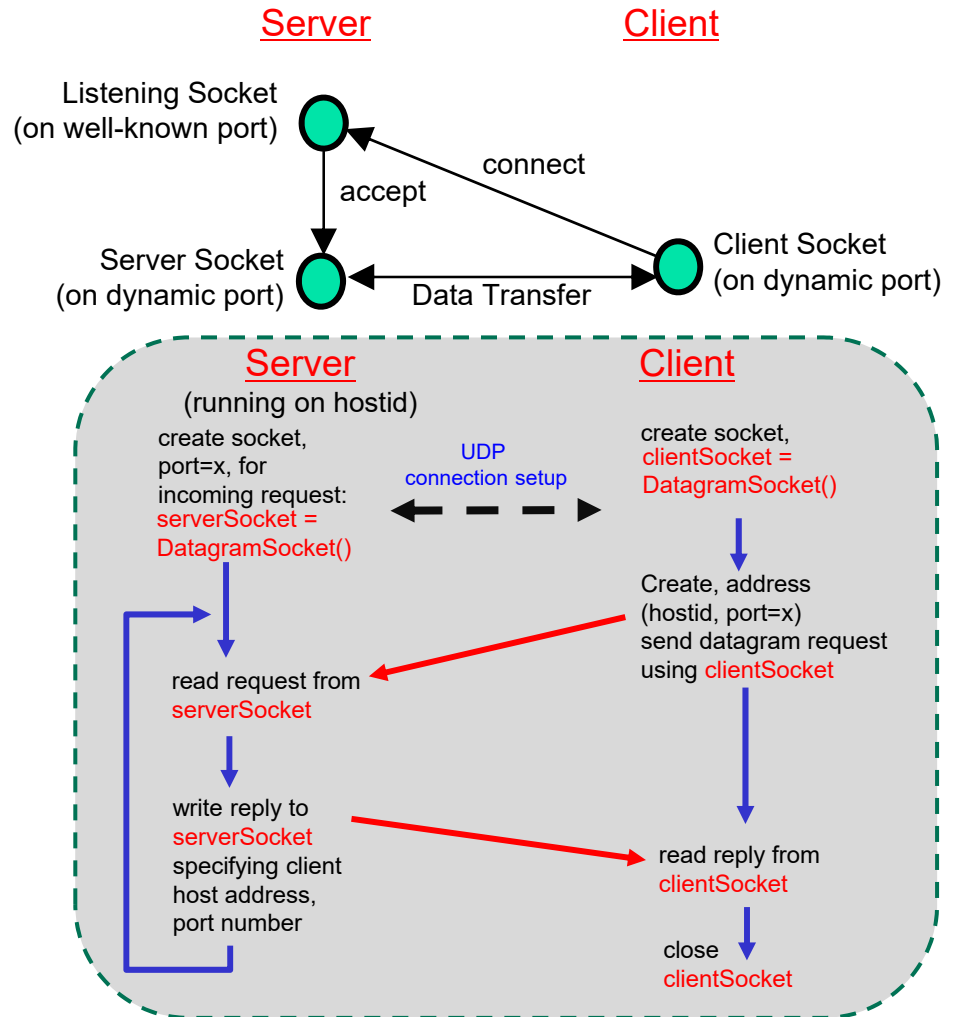


Figure: Client-Server Interaction

JAVA Example: Server, UDP

```
import java.io.*;
import java.net.*;
class UDPServer {
    public static void main(String args[]) throws Exception
    {
        DatagramSocket serverSocket = new DatagramSocket(9876);

        byte[] receiveData = new byte[1024];
        byte[] sendData = new byte[1024];

        while(true)
        {
            DatagramPacket receivePacket =
                new DatagramPacket(receiveData, receiveData.length);

            serverSocket.receive(receivePacket);

            String sentence = new String(receivePacket.getData());

            InetAddress IPAddress = receivePacket.getAddress();

            int port = receivePacket.getPort();

            String capitalizedSentence = sentence.toUpperCase();

            sendData = capitalizedSentence.getBytes();

            DatagramPacket sendPacket =
                new DatagramPacket(sendData, sendData.length, IPAddress, port);

            serverSocket.send(sendPacket);
        }
    }
}
```

Create datagram socket at port 9876

Create space for received datagram

Receive datagram


Get IP addr port #, of sender

Create datagram to send to client

Write out Datagram to socket

End of while loop, loop back and wait for another datagram

JAVA Example: Client, UDP

```
import java.io.*;
import java.net.*;
class UDPClient {
    public static void main(String args[]) throws Exception
    {
        
        BufferedReader inFromUser =
            new BufferedReader(new InputStreamReader(System.in));

        DatagramSocket clientSocket = new DatagramSocket();

        InetAddress IPAddress = InetAddress.getByName("hostname");

        byte[] sendData = new byte[1024];
        byte[] receiveData = new byte[1024];

        String sentence = inFromUser.readLine();
        sendData = sentence.getBytes();

        DatagramPacket sendPacket =
            new DatagramPacket(sendData, sendData.length, IPAddress, 9876);

        clientSocket.send(sendPacket);

        DatagramPacket receivePacket =
            new DatagramPacket(receiveData, receiveData.length);

        clientSocket.receive(receivePacket);

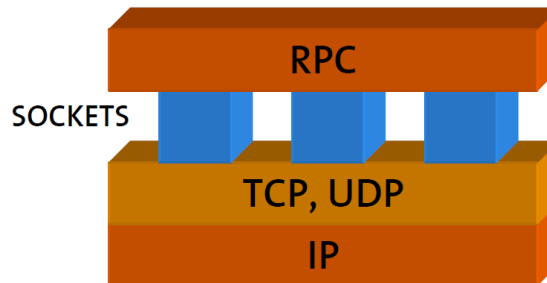
        String modifiedSentence =
            new String(receivePacket.getData());

        System.out.println("FROM SERVER:" + modifiedSentence);
        clientSocket.close();
    }
}
```

Socket Programming: Summary

Advantages	Disadvantages
<ul style="list-style-type: none">● Great control in the communication● It can be optimized as much as possible	<ul style="list-style-type: none">● Difficult to write applications● Every single detail in the communication has to be programmed

RPC Relation to TCP, UDP, IP



- TCP/IP and UDP/IP are visible to applications through sockets
- Purpose of the socket interface is to provide a UNIX-like file abstraction
- Yet sockets are quite low level for many applications, thus, RPC (**Remote Procedure Call**) appeared as a way to
 - ▣ Hide communication details behind a procedural call
 - ▣ Bridge heterogeneous environments
- RPC is a standard for distributed computing (client-server)

Remote Procedure Call (RPC)

- Interaction between client and server is abstracted as a **procedure call**
- **Server** exports a set of procedures that the client use remotely
- **Client** uses these remote procedures as if they are local
- All the complexity of the network is hidden
- Programmer only concentrates in writing the procedures and using them

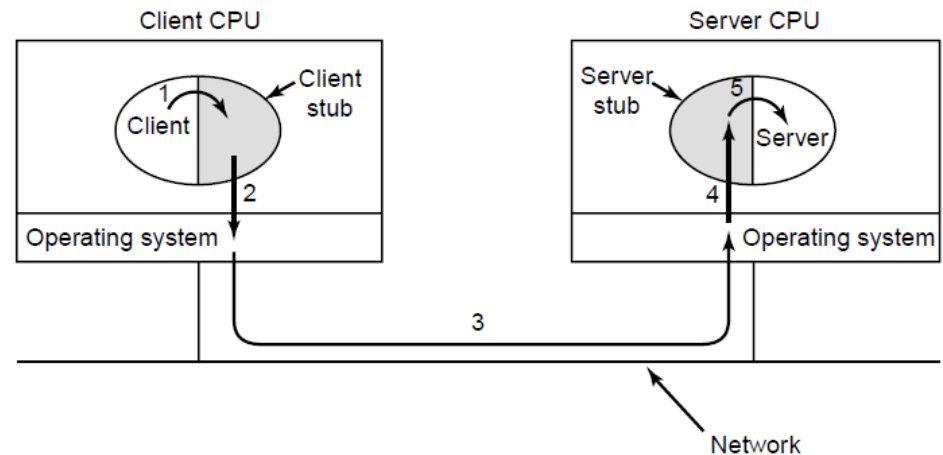


Figure: Steps in making a remote procedure call. The stubs are shaded.

RPC Tools

- Programmer describes the procedures' interface in an **Interface Definition Language (IDL)**
- Using a RPC preprocessor, the IDL description is transformed in
 - ▣ **Server code** contains the dispatcher loop and all the code needed to call to get the arguments from the client, call the procedures in the server and return the results
 - ▣ **Client code** contains “**Stub**” procedures that the client code call. These stubs send the arguments to the server and receive the results

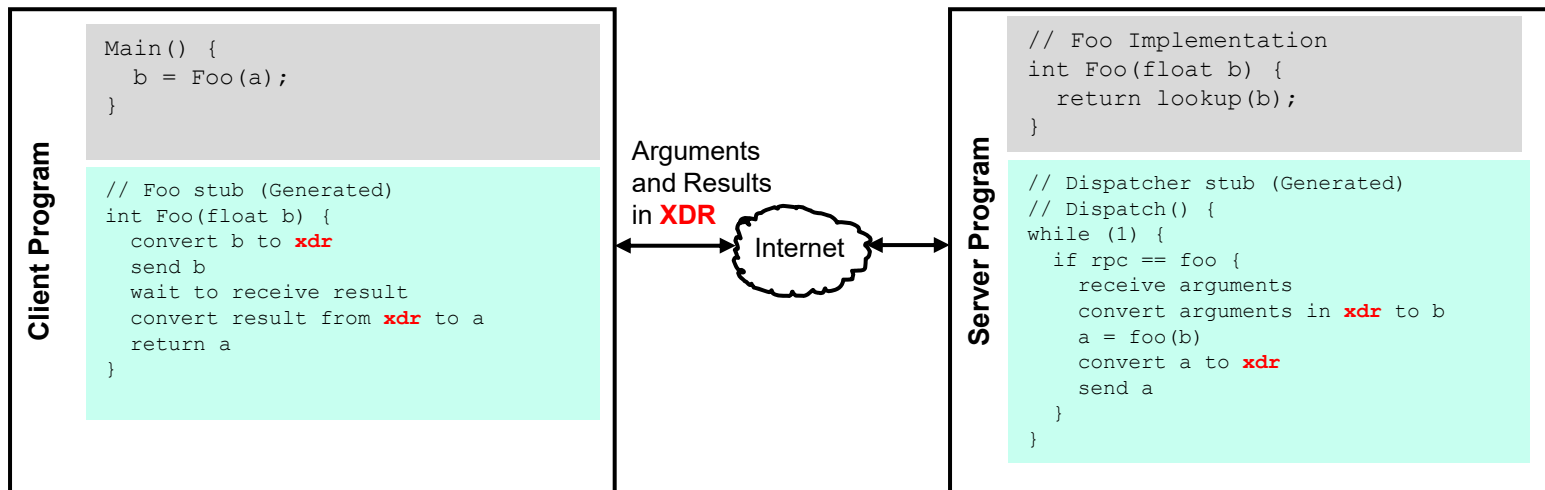


Figure: RPC Architecture

XDR: External Data Representation

- XDR is used to allow RPC's across machines that have different data representations
- Arguments and Results are converted to XDR before they cross the Internet
- XDR is a canonical representation used to send data in the Internet
- Client/Server code needs to convert the data from the host representation to XDR before sending and back when received
- RPC processor generates the XDR filters from the IDL representation
- XDR: RFC1014

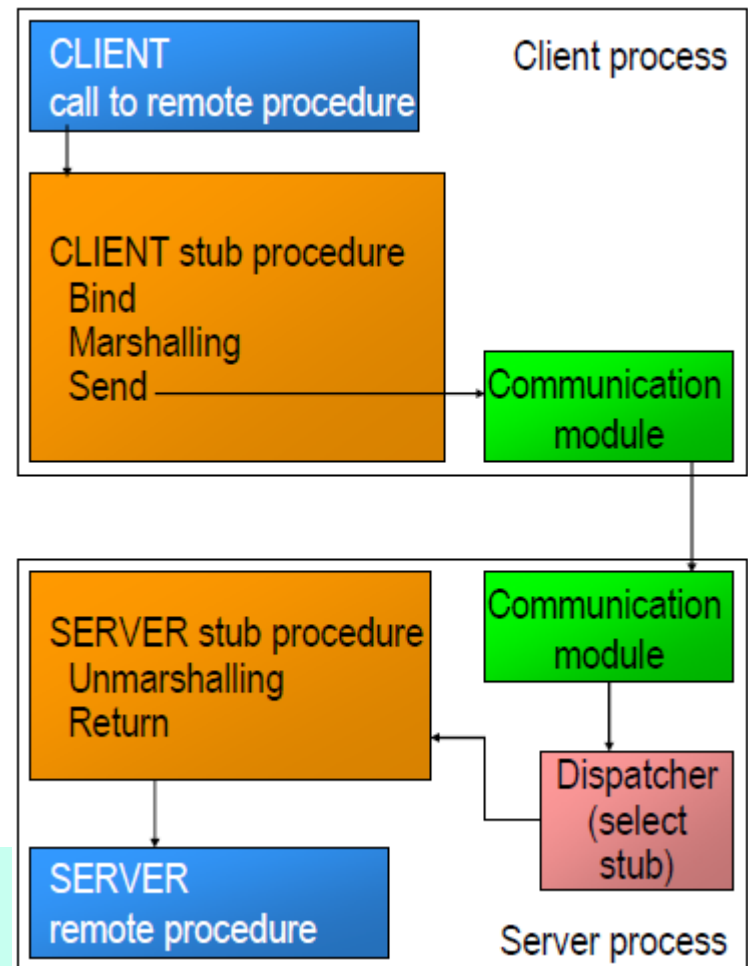
Using RPCs

- Programmer
 - Writes the IDL description of the RPC's
 - Implements the procedures
 - Uses the Remote procedures as if they are local
 - Runs the RPC preprocessor on the IDL description of the RPC's
- RPC preprocessor
 - Generates stubs for the client program
 - Generates the dispatcher for the server program
 - Generates the XDR filters that do the conversion
- RPC implementation
 - SunRPC
 - DCE RPC
 - OMG CORBA
 - Windows COM
 - HORB, Java RMI
 - XML-RPC
 - SOAP

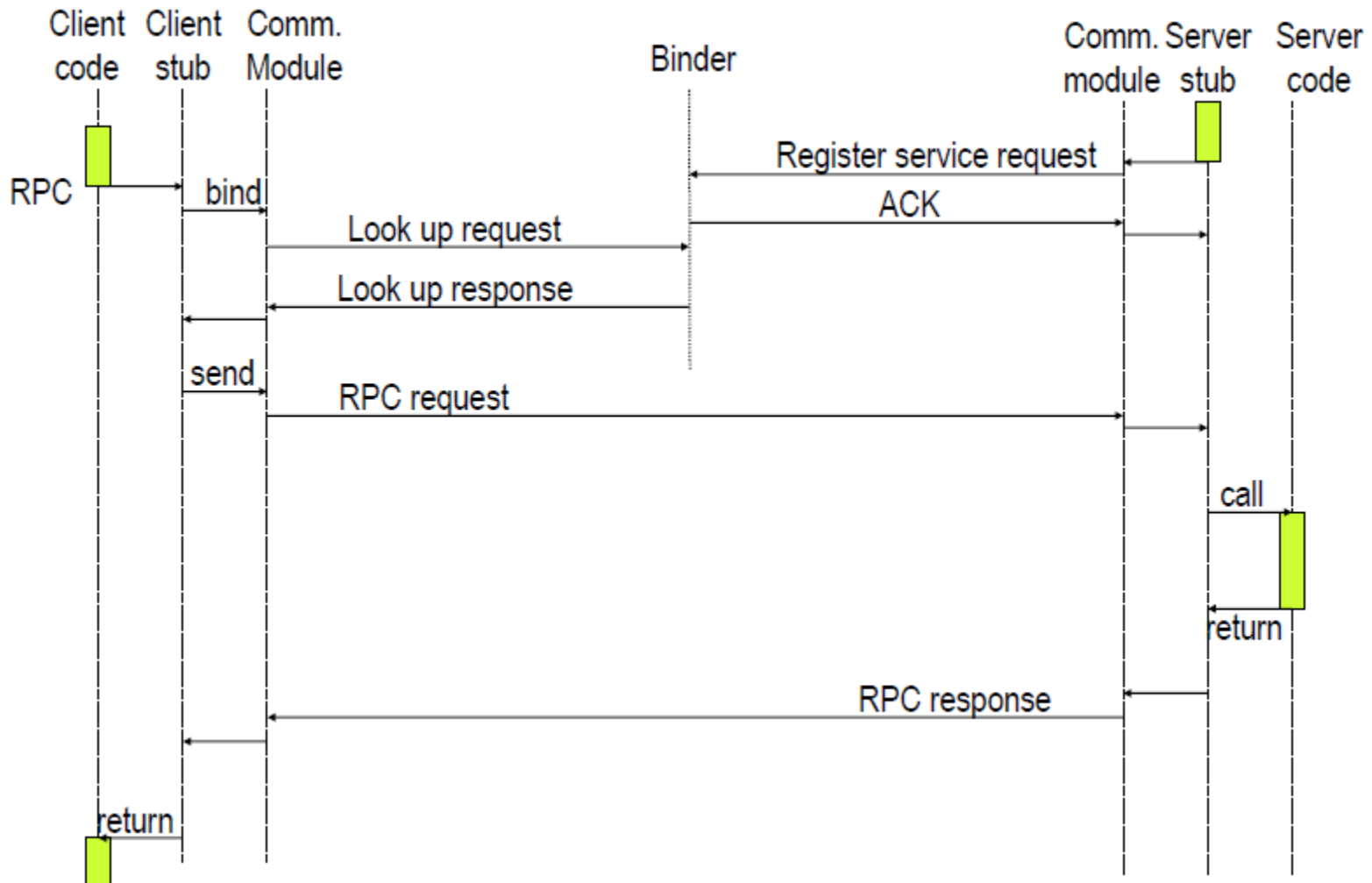
Making RPC Work in Practice

- One cannot expect the programmer to implement all these mechanisms every time a distributed application is developed. Instead, they are provided by RPC system (example of low level middleware)
- What does an RPC system do?
 - ▣ Provides an IDL to describe the services
 - ▣ Generates all the additional code necessary to make a procedure call remote and to deal with all the communication aspects
 - ▣ Provides a binder in case it has a distributed name and directory service system

- **Client stub** packs the parameters into a message and makes a system call to send the message. Packing the parameters is called **marshalling**
- **server stub** unpacks the parameters from the message. Unpacking the parameters is called **unmarshalling**



More Details



RPC in Pseudo Code

IDL Description

```
//your client code
result = function(parameters)
```



```
//client side stub
function(parameters) {
  address a = bind("function");
  socket s = connect(a);
  send(s,"function");
  send(s,parameters);
  receive(s,result); //blocking
  return result;
}
```

```
//rpc server main loop
void rpc_server() {
  register("function",address);
  while (true) {
    socket s = accept(); //blocking
    receive(s,id);
    if (id == "function")
      dispatch_function(s);
    close(s);
  }
}
```

```
//server side stub
void dispatch_function(socket s) {
  receive(s,parameters);
  result = function(parameters);
  send(s,result);
}
```


Sun RPC

- First successful RPC implementation
- It used to program Network File System (NFS) and Yellow Pages
- Public Domain
- Preprocessor is called **rpcgen**, which is now part of the UNIX distribution
- Sun RPC has only 3 calls:
 - **rpc_reg()** registers a procedure as a remote procedure and returns a unique, system-wide identifier for the procedure
 - **rpc_call()** given a procedure identifier and a host, it makes a call to that procedure
 - **rpc_broadcast()** is similar to **rpc_call()** but broadcasts the message instead
- IDL compiler automatically generates the stubs calling the RPC library using defaults
- Direct access allow more control of transport protocols, marshalling, binding procedures, etc

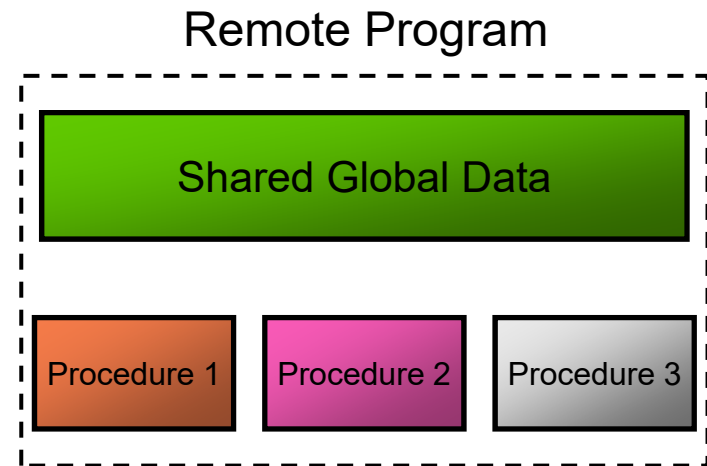
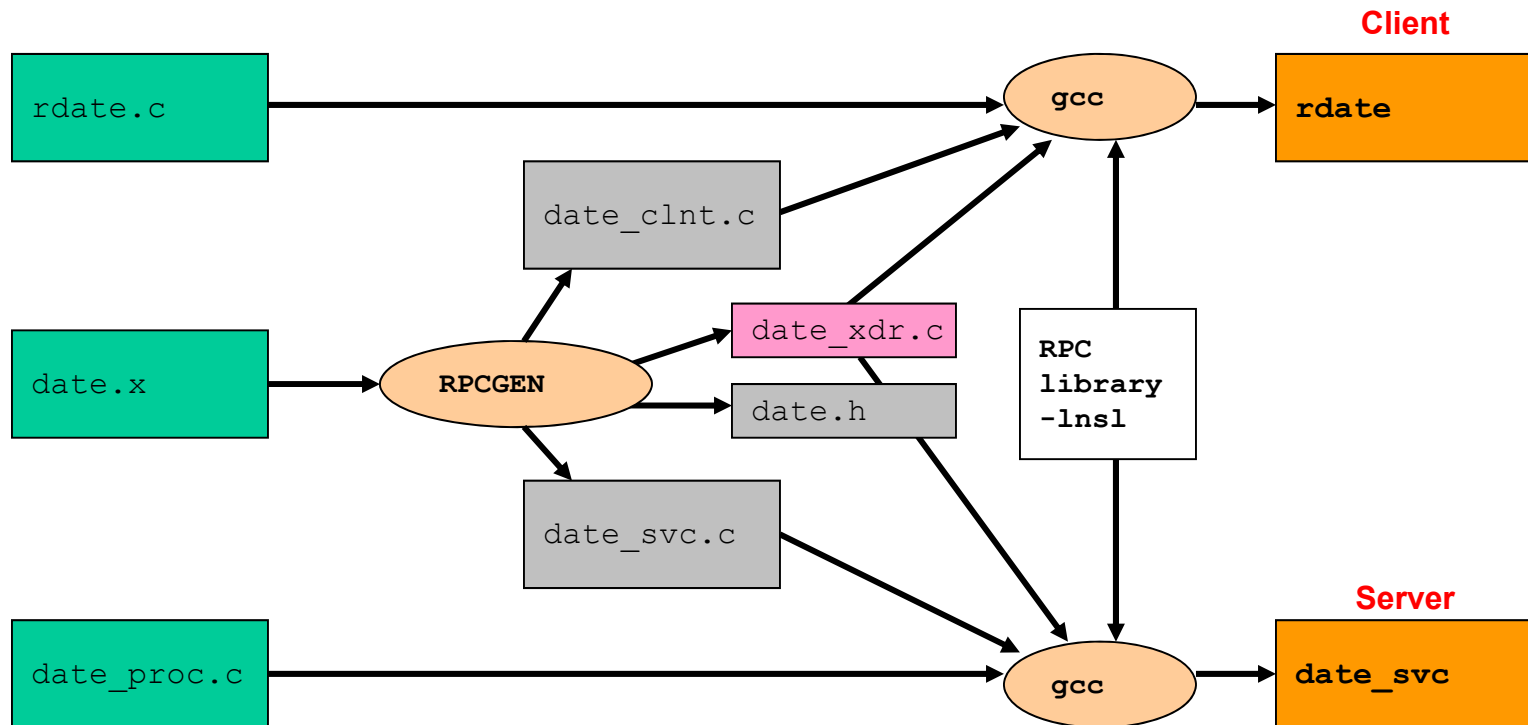


Figure: Sun RPC organization

- Procedure Arguments
 - To reduce the complexity of the interface specification, Sun RPC includes support for a single argument to a remote procedure*
 - Typically the single argument is a structure that contains a number of values

* Newer versions can handle multiple args

Sun RPC Example



- `rdate` program from W.R. Stevens, UNIX Network Programming (RPCsample.tar is available online). Original files are `date.x`, `rdate.c`, `date_proc.c`
- Generate stubs
- Compile & link the client and client stub
- Compile & link the server and server stub

```
rpcgen date.x
```

```
cc -o rdate rdate.c date_clnt.c -lnsl
```

```
cc -o date_svc date_proc.c date_svc.c -lnsl
```

RPC: Summary

Advantages	Disadvantages
<ul style="list-style-type: none"> ● RPC provides a mechanism to implement distributed applications in a simple and efficient manner ● RPC follows the programming techniques of the time (procedural languages) and fitted quite well with the most typical programming languages (C), thereby facilitating its adoption by system designers ● RPC allows the modular and hierarchical design of large distributed systems: <ol style="list-style-type: none"> ① Client and server are separate ② Server encapsulates and hides the details of the back end systems (such as databases) 	<ul style="list-style-type: none"> ● RPC is not a standard, it is an idea that has been implemented in many different ways (not necessarily compatible) ● RPC allows designers to build distributed systems but does not solve many of the problems distribution creates. In that regard, it is only a low level construct ● RPC is designed with only one type of interaction in mind: client/server. This reflects the hardware architectures at the time when distribution meant small terminals connected to a mainframe. As hardware and networks evolve, more flexibility is needed

Announcement

- **Mid-term Examination**
 - 09:00 ~ 10:40 on 7 November (Monday)

- Chapter 8 Name Service and Internetworking
 - 13:30 ~ 15:10 on 7 November (Monday)