School of Computing Science
Simon Fraser University

CMPT 471: Computer Networking II

Introduction

Instructor: Mohamed Hefeeda

Course Objectives

- Understand
  - principles of designing and operating computer networks
  - structure and protocols of the Internet
  - services that can/cannot be offered by the Internet
- Know how to
  - implement network protocols and applications
- Be informed about
  - recent/hot topics in networking research and industry
  - top technical conferences/journals in networking research

Course Info: Textbooks and References

- **Textbooks**
  - Kurose and Rose, Computer Networking: A top-down Approach, 7th edition, 2016
  - Peterson and Davie, Computer Networks: A Systems Approach,, 5th edition, 2012.
    - Available Online through SFU Library.
- **References**
  - Posted on the course web page
- **Course web page**
  - Check: courses.cs.sfu.ca

Course Info: Grading (Tentative)

- **Assignments, Projects: 50%**
  - Several programming projects, mostly in C and Java
  - Assignments may include problems sets, researching topics, conducting experiments, presentations, …
  - Must read Assignment Policy (on course web page)
- **Exams: 50%**

Course Info: Topics

- Review of Networking Basics:
  - Internet Architecture and TCP/IP Stack
- IP Multicast
- Multimedia Networking
- Wireless Networks
- Selected topics from
  - Virtual Networks and Overlays
  - Network Security
  - Software Defined Networks
  - Cloud Computing
  - Data Center Networking

Quick Survey: did you cover …

- Socket programming?
- Wireshark experiments?
- IP Multicast?
- Multimedia Networking?
- Wireless Networks?
- Network Security?

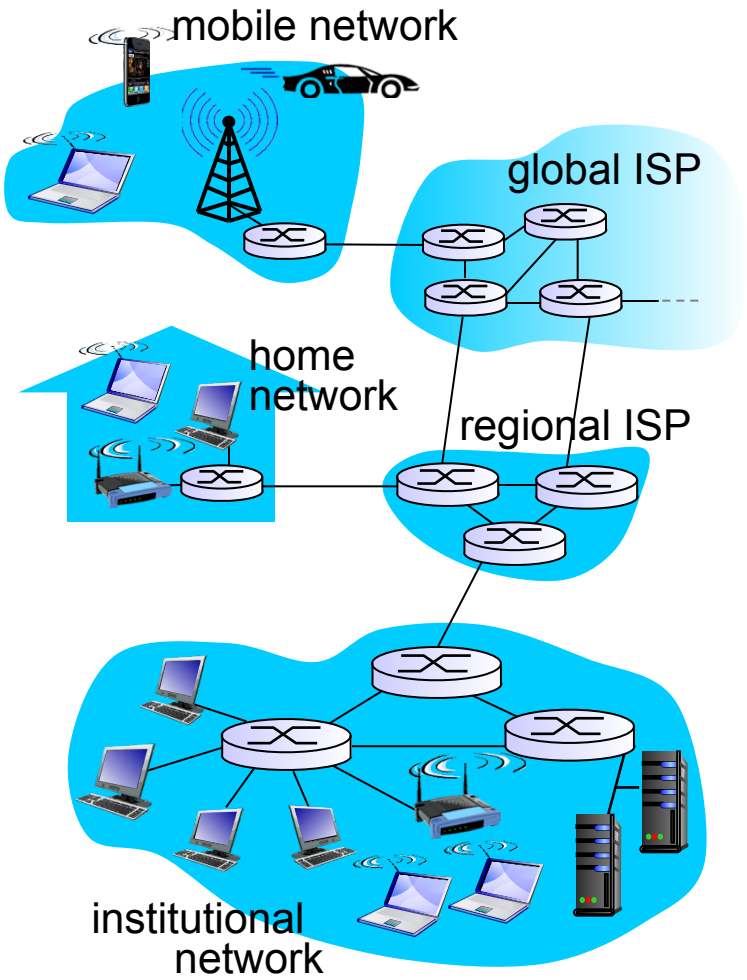
Basic Networking Concepts

Review of Basic Networking Concepts

- Internet structure
- Protocol layering and encapsulation
- Socket programming
- Transport layer
  - Reliability and congestion control
  - Performance modeling of TCP
- Network Layer
  - Addressing, Forwarding, Routing
  - IP Multicast

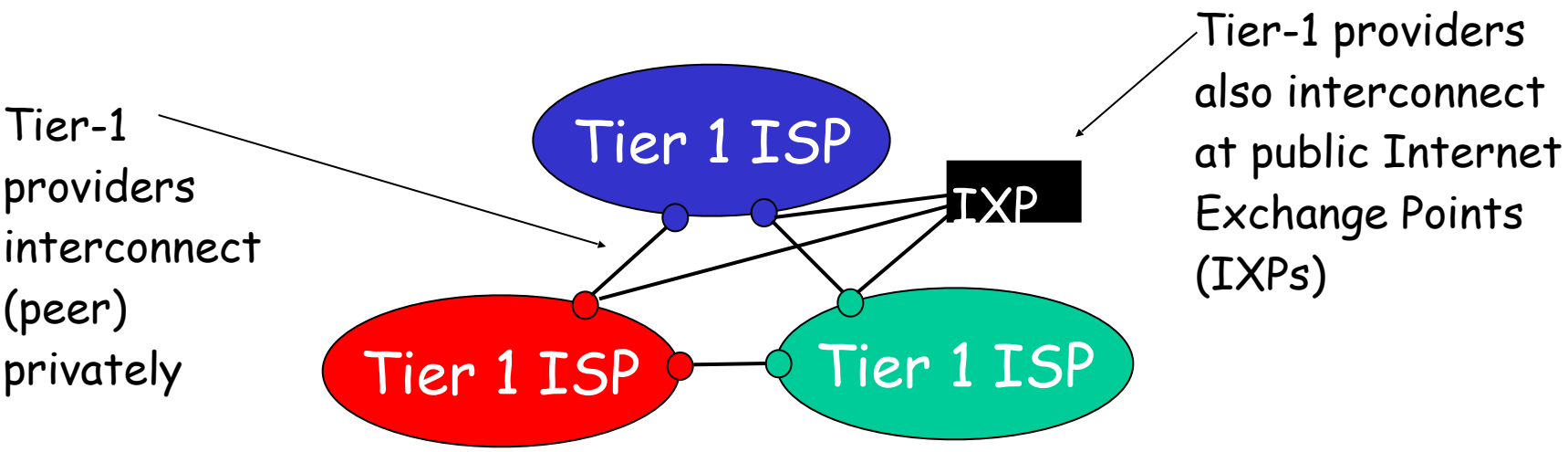
The Internet: Network of Networks

- *Internet:* "network of networks"
  - Interconnected ISPs
- *protocols* control sending, receiving of messages
  - e.g., TCP, IP, HTTP, Skype, 802.11
- *Internet standards*
  - RFC: Request for comments
  - IETF: Internet Engineering Task Force

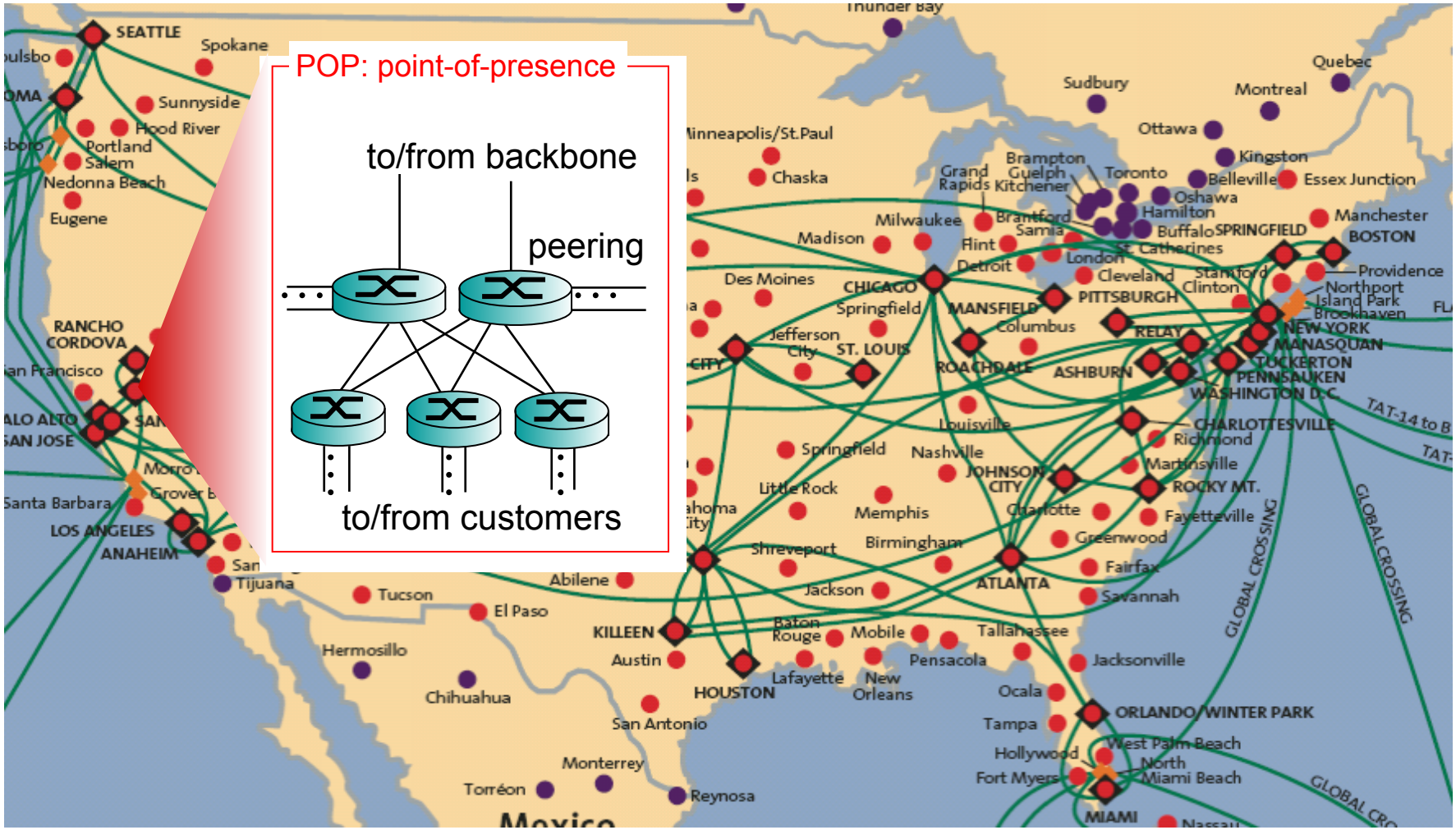


Internet structure: network of networks

- roughly hierarchical
- at center: "tier-1" ISPs (e.g., MCI, Sprint, and AT&T), national/ international coverage
  - treat each other as equals

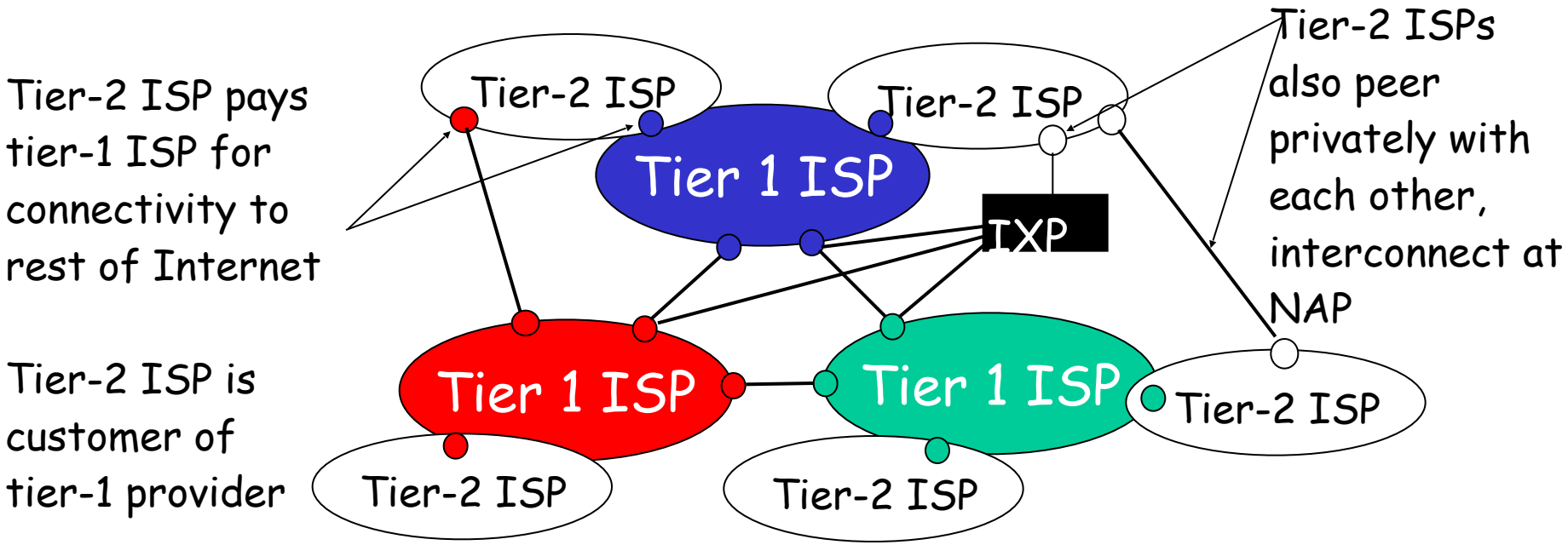


Tier-1 ISP: e.g., Sprint



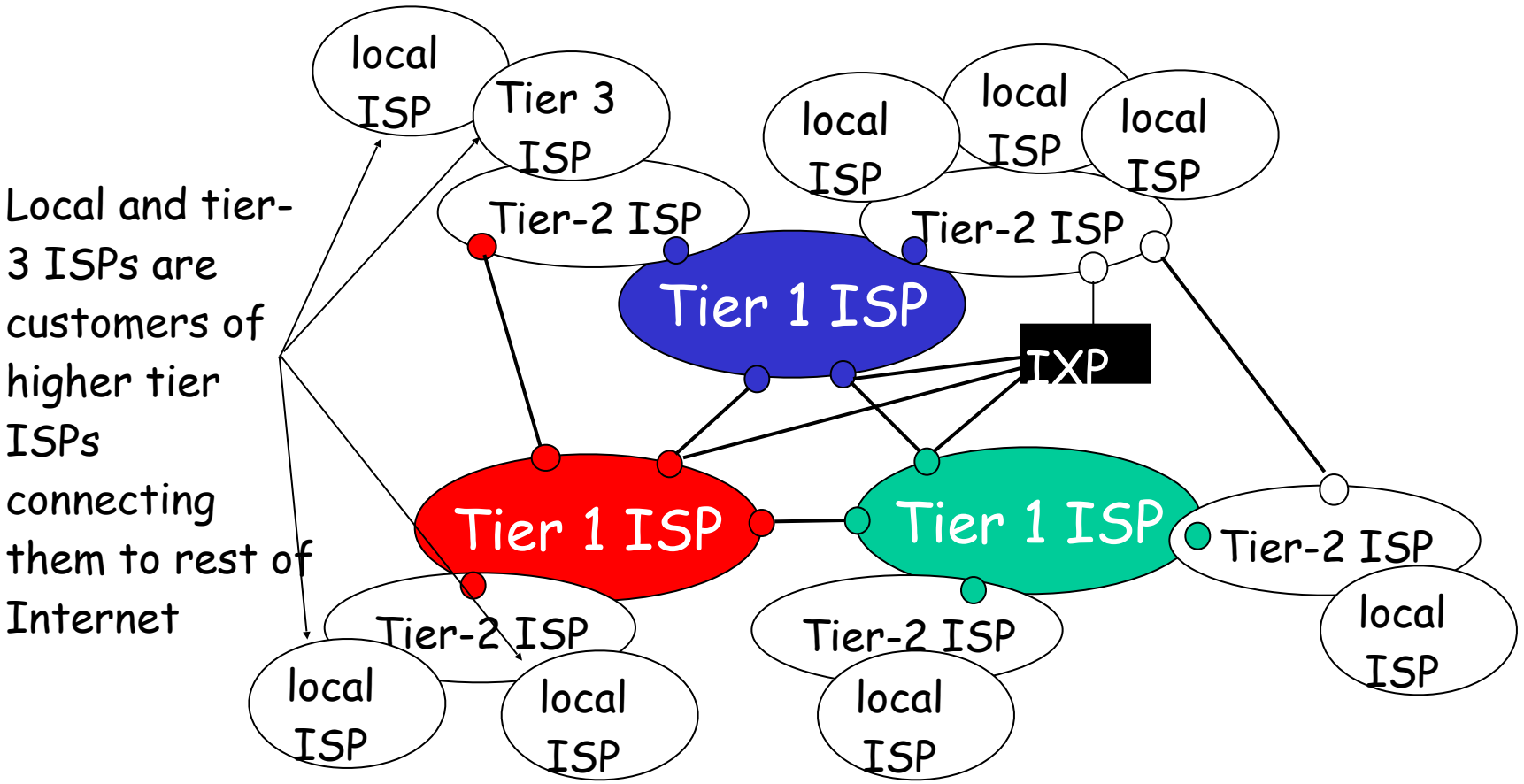
Internet structure: Tier-2 ISPs

- "Tier-2" ISPs: smaller (often regional) ISPs
  - Connect to one or more tier-1 ISPs, possibly other tier-2 ISPs



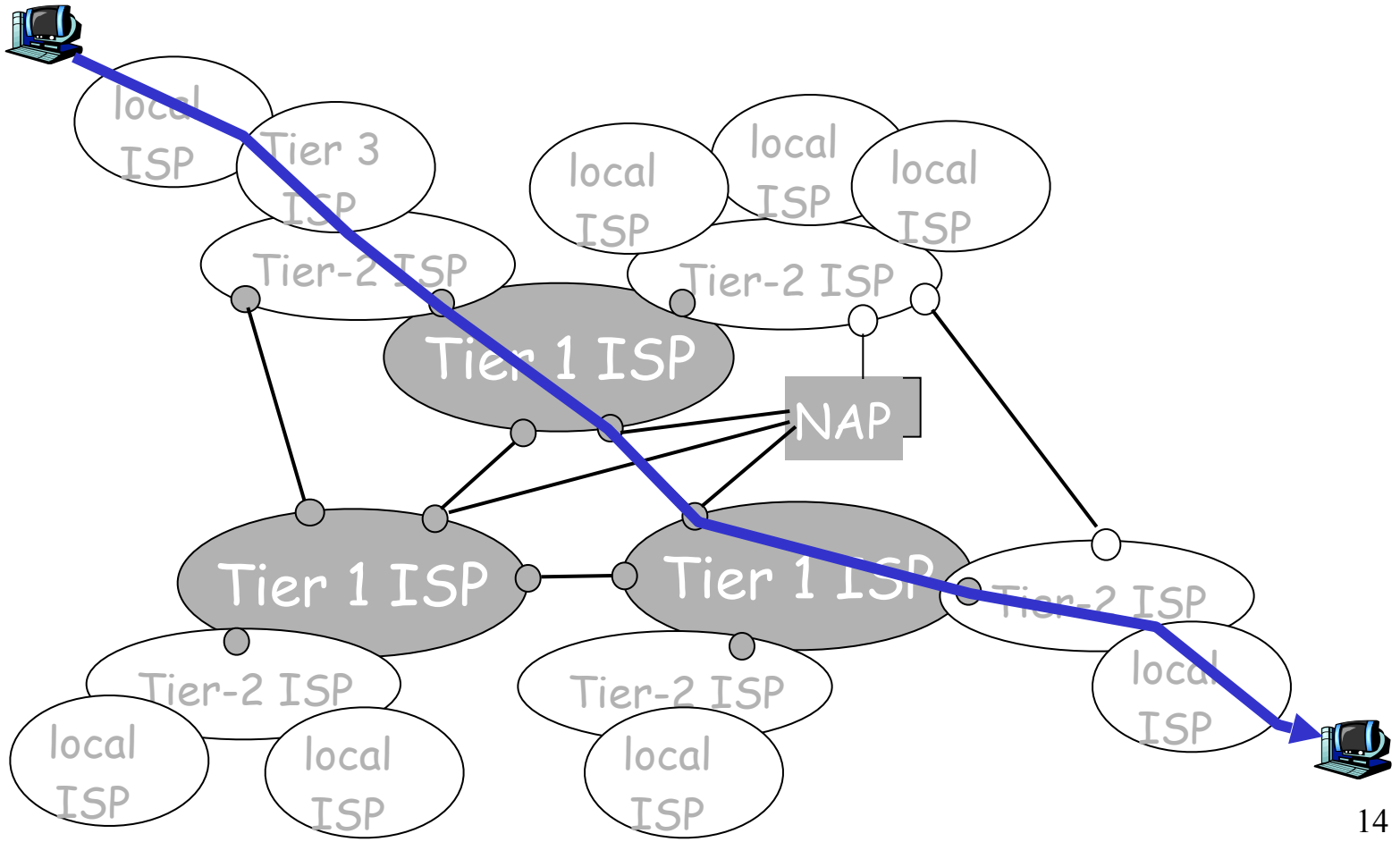
Internet structure: Tier-3 ISPs

- "Tier-3" ISPs and local ISPs
  - last hop ("access") network (closest to end systems)



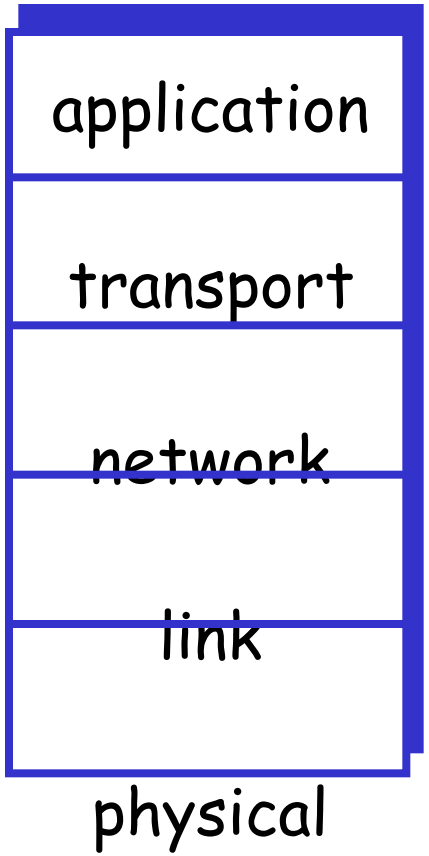
Internet structure: packet journey

- a packet passes through many networks!

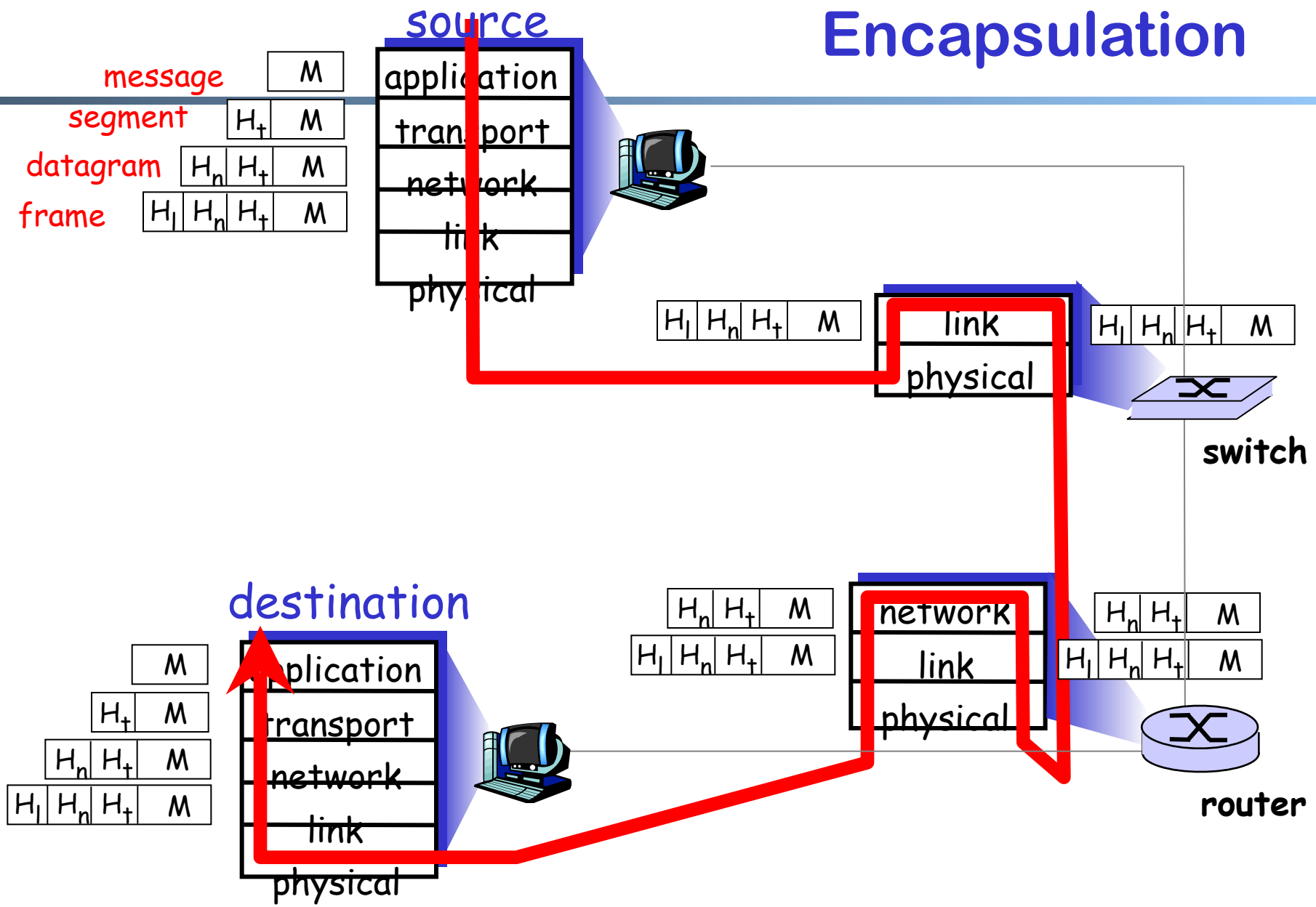


Internet protocol stack

- **application:** supporting network applications
  - FTP, SMTP, HTTP
- **transport:** process-process data transfer
  - TCP, UDP
- **network:** routing of datagrams from source to destination
  - IP, routing protocols
- **link:** data transfer between neighboring network elements
  - PPP, Ethernet
- **physical:** bits "on the wire"



Encapsulation



Internet Services

- View the Internet as a communication infrastructure that provides services to apps
  - Web, email, games, e-commerce, file sharing, …
- Two communication services
  - Connectionless unreliable
  - Connection-oriented reliable

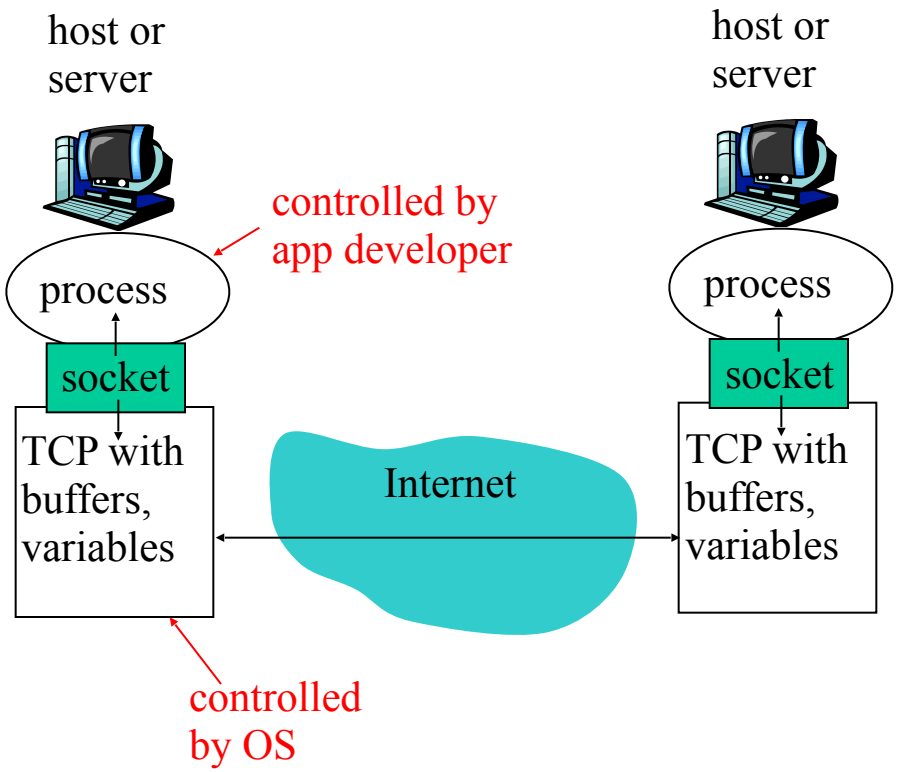
Internet Services

- Connection-oriented
  - Prepare for data transfer ahead of time
  - establish connection → set up state in the two communicating hosts
  - Usually comes with reliability, flow and congestion control
  - TCP: Transmission Control Protocol
- Connectionless
  - No connection set up, simply send
  - Faster, less overhead
  - No reliability, flow control, or congestion control
  - UDP: User Datagram Protocol

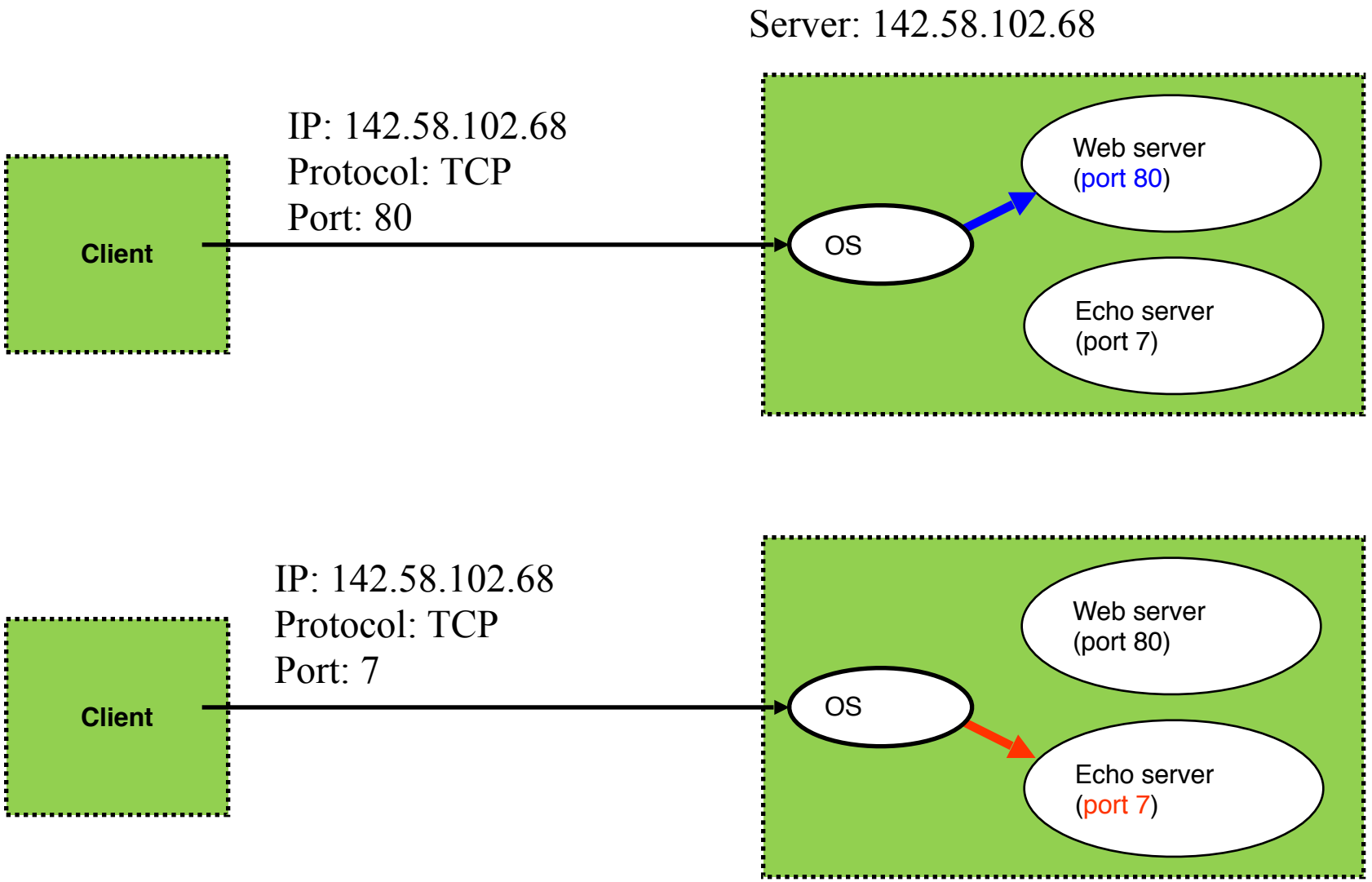
How can we access these services?

Network (Socket) Programming

- Process sends/receives messages to/from its socket
- Socket is the interface (API) between application and transport layer
- Process is identified by:
  - IP address,
  - Transport protocol, and
  - Port number



Identifying Processes



Port Numbers

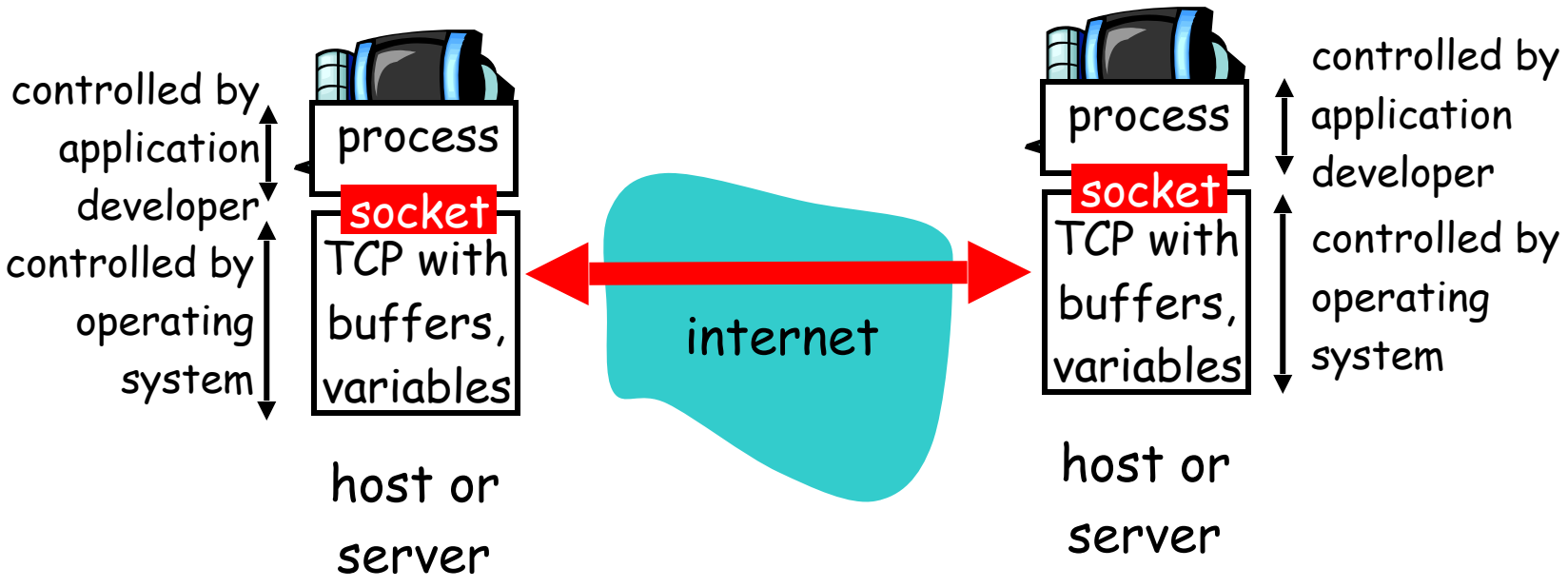
- Popular applications have well-known ports
  - E.g., port 80 for Web and port 25 for e-mail
  - See http://www.iana.org/assignments/port-numbers
- Server port:
  - Known/fixed (e.g., port 80)
  - Ports between 0 and 1023 (require root to use)
- Client port:
  - Client chooses an unused ephemeral (i.e., temporary) port, Between 1024 and 65535

UNIX Socket API

- Socket interface
  - Originally provided in Berkeley UNIX
  - Later adopted by all popular operating systems
  - Simplifies porting applications to different OSes
- In UNIX, everything is like a file
  - All input is like reading a file
  - All output is like writing a file
  - File is represented by an integer file descriptor
- API implemented as system calls
  - E.g., connect, read, write, close, …

Socket Programming using TCP

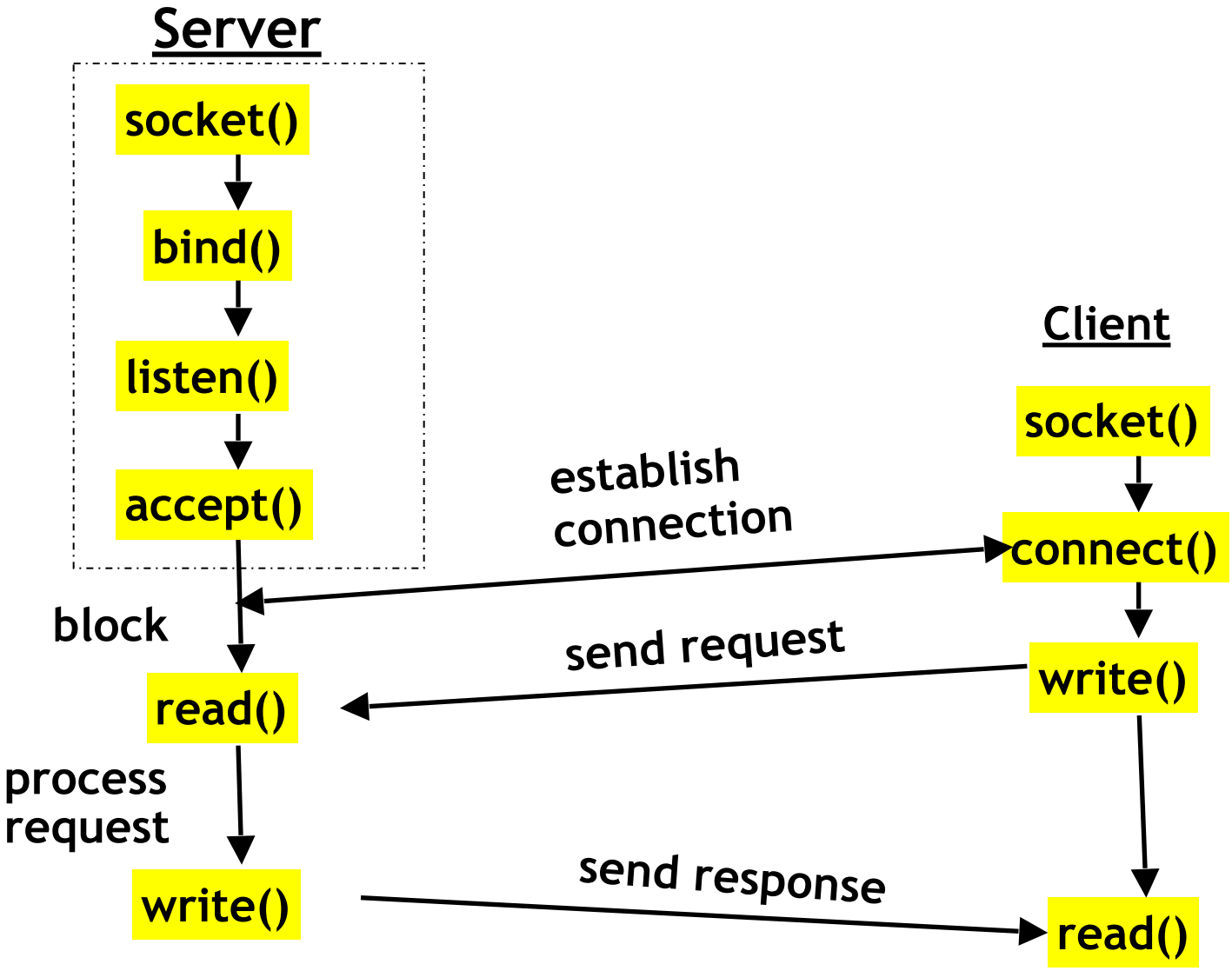
- **TCP service:** reliable transfer of **bytes** from one process to another
  - virtual pipe between sender and receiver



Socket Programming using TCP

- Server process must be running first, and
  - creates a socket (door) that accepts client's contact, then wait
- Client contacts server by creating local TCP socket using IP address, port number of server process
- When client creates socket
  - client TCP (in OS kernel) establishes connection to server TCP
  - Then data start to flow

TCP Socket: Basic Structure (Unix/C)

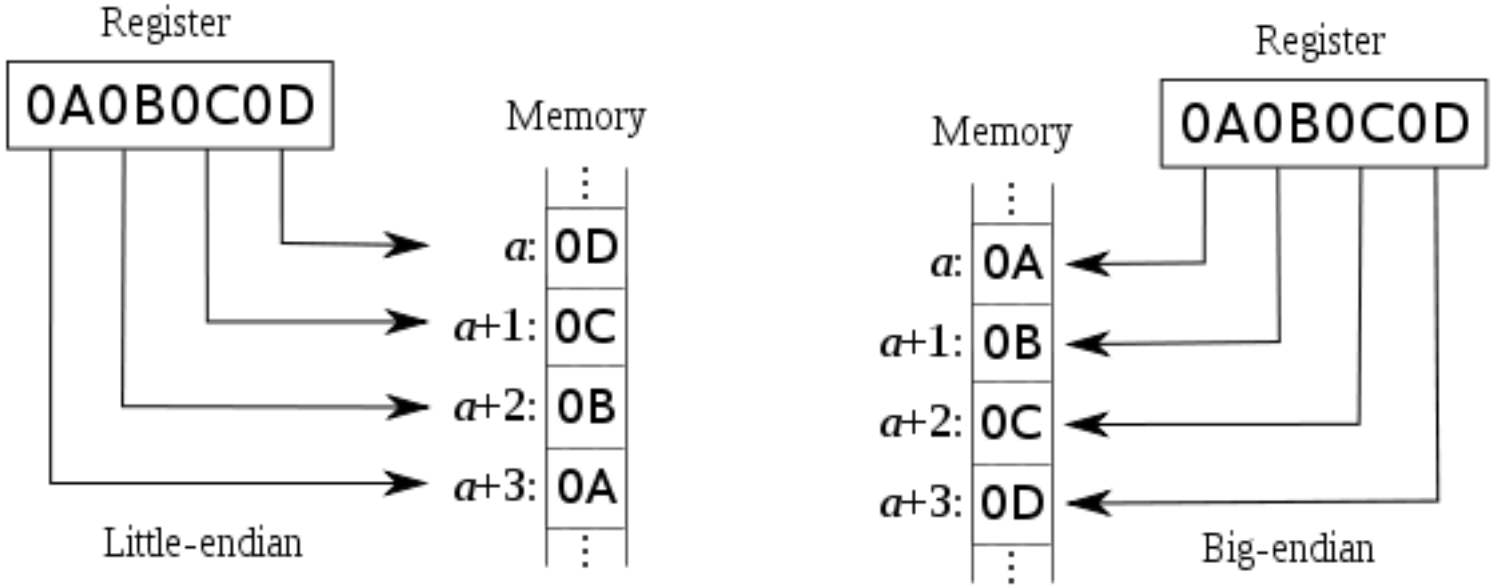


TCP Daytime Server

```
int main (int argc, char **argv) {
        int listenfd, connfd;
        struct sockaddr_in servaddr;
        char buff[MAXLINE];
        time_t ticks;
        listenfd = socket(AF_INET, SOCK_STREAM, 0);
        bzero(&servaddr, sizeof(servaddr));
```

htonX and ntohX macros: Important

- Some machines use "big endian" and others use "little endian" to store numbers
  - Whenever sending numbers to network use htonX
  - Whenever receiving numbers from network use ntohX
    - Replace X with l for long integer (4 bytes), and s for short (2 bytes)



Creating Socket: socket()

- `int socket(int domain, int type, int protocol)`
  - Returns a file descriptor (or handle) for the socket
- domain: protocol family
  - PF_INET for the Internet (IPv4)
- type: semantics of the communication
  - SOCK_STREAM: reliable byte stream (TCP)
  - SOCK_DGRAM: message-oriented service (UDP)
- protocol: specific protocol
  - UNSPEC: unspecified
  - (PF_INET and SOCK_STREAM already implies TCP)

TCP Daytime Client

```
int main(int argc, char **argv) {
 …
        if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
                printf("socket error\n"); exit(1); }
```

Concurrent TCP Servers

- Daytime server accepts one connection at a time
  - Not good for other servers, e.g., Web servers
- How would you make it handle multiple connections concurrently?
- We need some parallelism!
  - But where?

TCP Daytime Server

```
int main (int argc, char **argv) {
        int listenfd, connfd;
        struct sockaddr_in servaddr;
        char buff[MAXLINE];
        time_t ticks;
        listenfd = socket(AF_INET, SOCK_STREAM, 0);
        bzero(&servaddr, sizeof(servaddr));
        servaddr.sin_family = AF_INET;
        servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
        servaddr.sin_port = htons(DAYTIME_PORT); /* daytime server */
        bind(listenfd, (struct sockaddr *) &servaddr, sizeof(servaddr));
        listen(listenfd, LISTENQ);

        for ( ; ; ) {
                connfd = accept(listenfd, (struct sockaddr *) NULL, NULL);
                ticks = time(NULL);
                snprintf(buff, sizeof(buff), "%.24s\r\n", ctime(&
                write(connfd, buff, strlen(buff));
                printf("Sending response: %s", buff);
                close(connfd);
    }}
```

here

Concurrent Server

```
for ( ; ; ) {
        connfd = accept(listenfd, …);

        if ( (pid = fork()) == 0) {
                close(listenfd); /*child closes listening socket */

                doit(connfd); /*process the request */

                close(connfd); /*done with this client */

                exit(0); /*child terminates */
        }

        close(connfd); /*parent closes connected socket */
}
```

- Fork: duplicates the entire process that called it
- Fork returns twice!
  - One to the child process, return value = 0
  - Second to parent with non zero (pid of the created child)

Summary

- Internet structure
- Protocol layering
- Socket programming