CS183

Instructor: Ali Davanian

Most slides were adopted from Romit Roy Choudhury and and Ishani Janveja (from UC Berkeley)

L4 Networks and Sockets

What to learn from Transport Layer?

- Understand principles behind transport layer services:
  - Multiplexing/demultiplexing, reliable data transfer, flow control, congestion control
- Learn about transport layer protocols in the Internet:
  - UDP: connectionless transport
  - TCP: connection-oriented transport
- Learn Socket Programming in Linux
  - Sometimes troubleshooting network services require deep understanding of how sockets work

Transport vs. network layer

- *Network layer:* logical communication between hosts
- *Transport layer:* logical communication between processes
  - relies on, enhances, network layer services

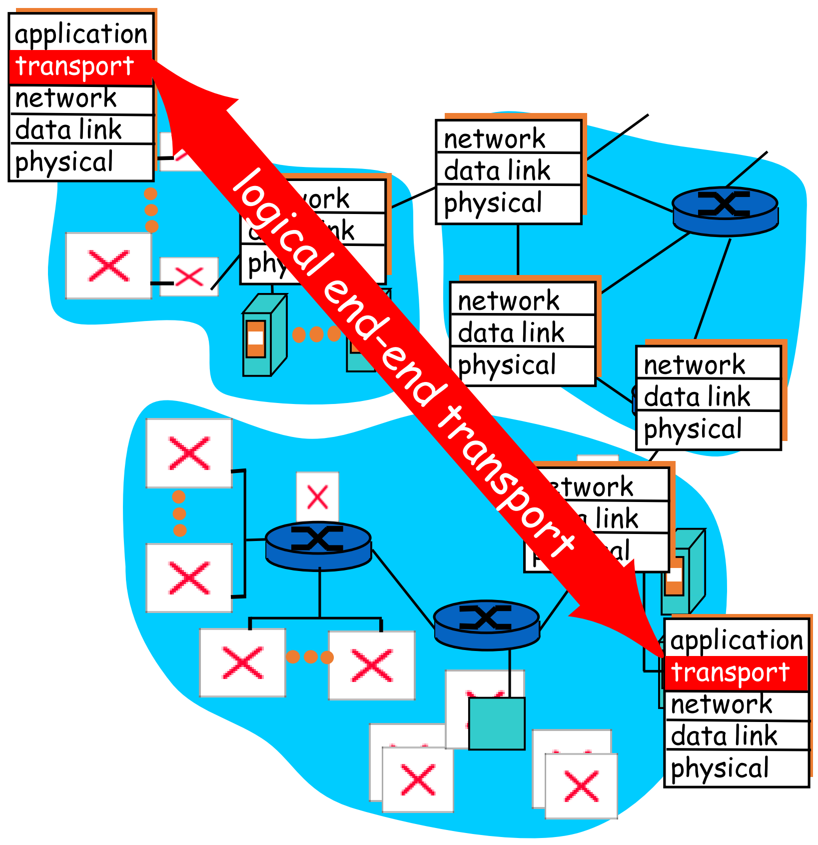
<u>Household analogy:</u>

12 kids sending letters to 12 kids

- processes = kids
- app messages = letters in envelopes
- hosts = houses
- transport protocol = Ann to Bill
- network-layer protocol = postal service

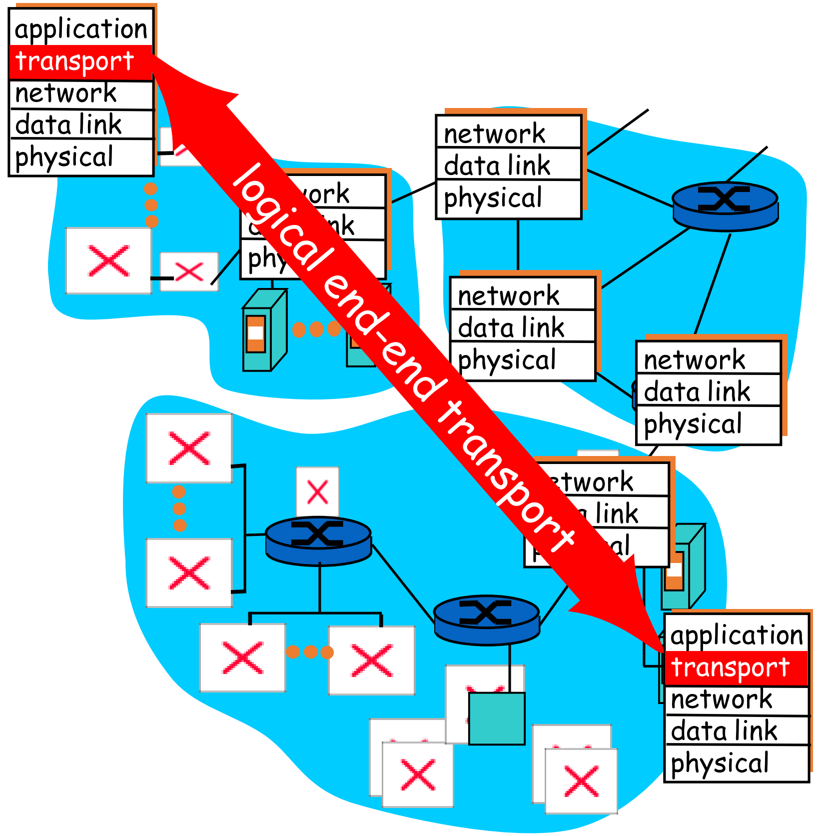
Transport services and protocols

- Provide logical communication between app processes running on different hosts
- Transport protocols run in end systems
  - Sender: breaks app messages into segments, passes to network layer
  - receiver: reassembles segments into messages, passes to app layer
- More than one transport protocol available to apps
  - Internet: TCP and UDP

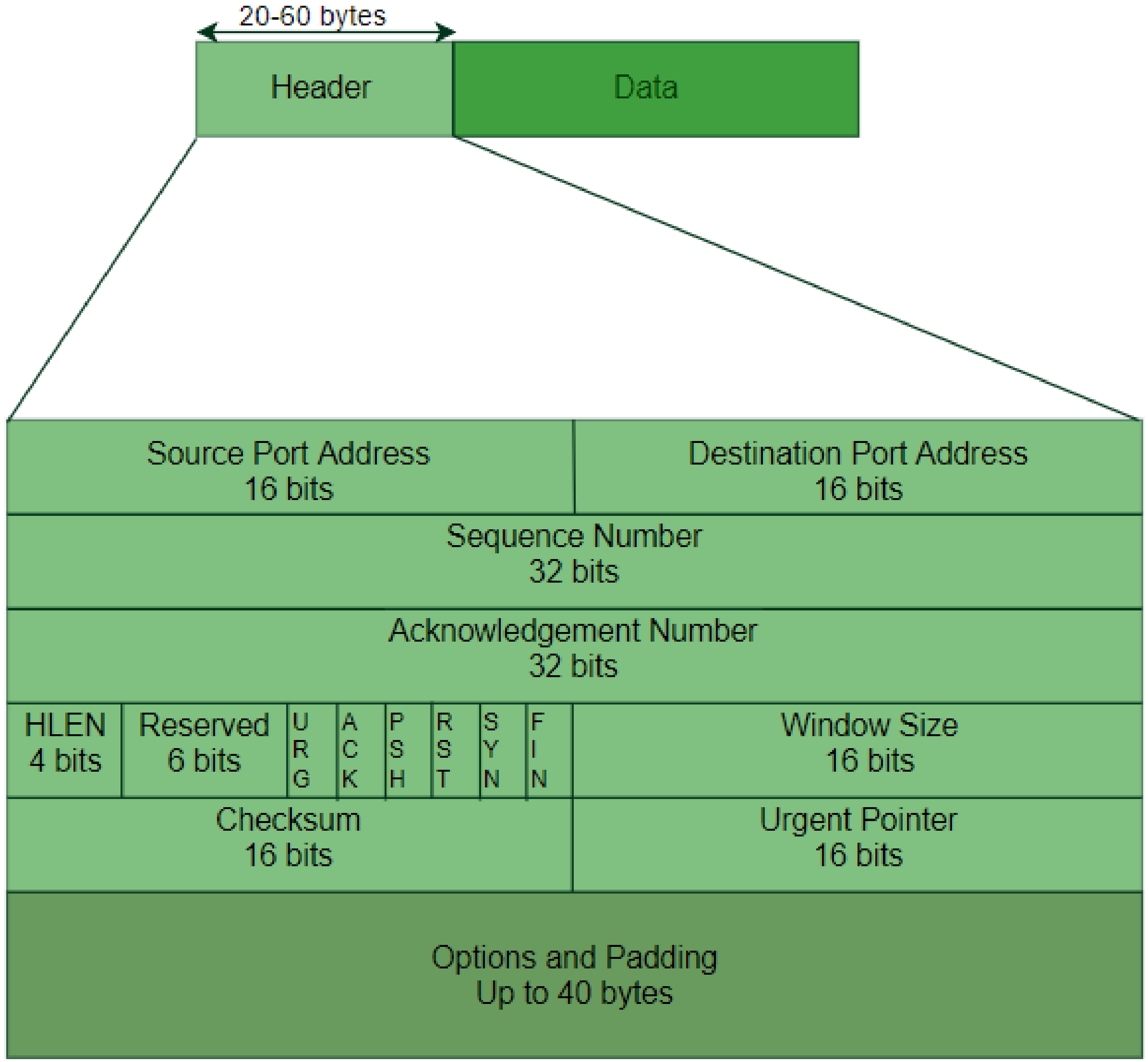


Transport services and protocols

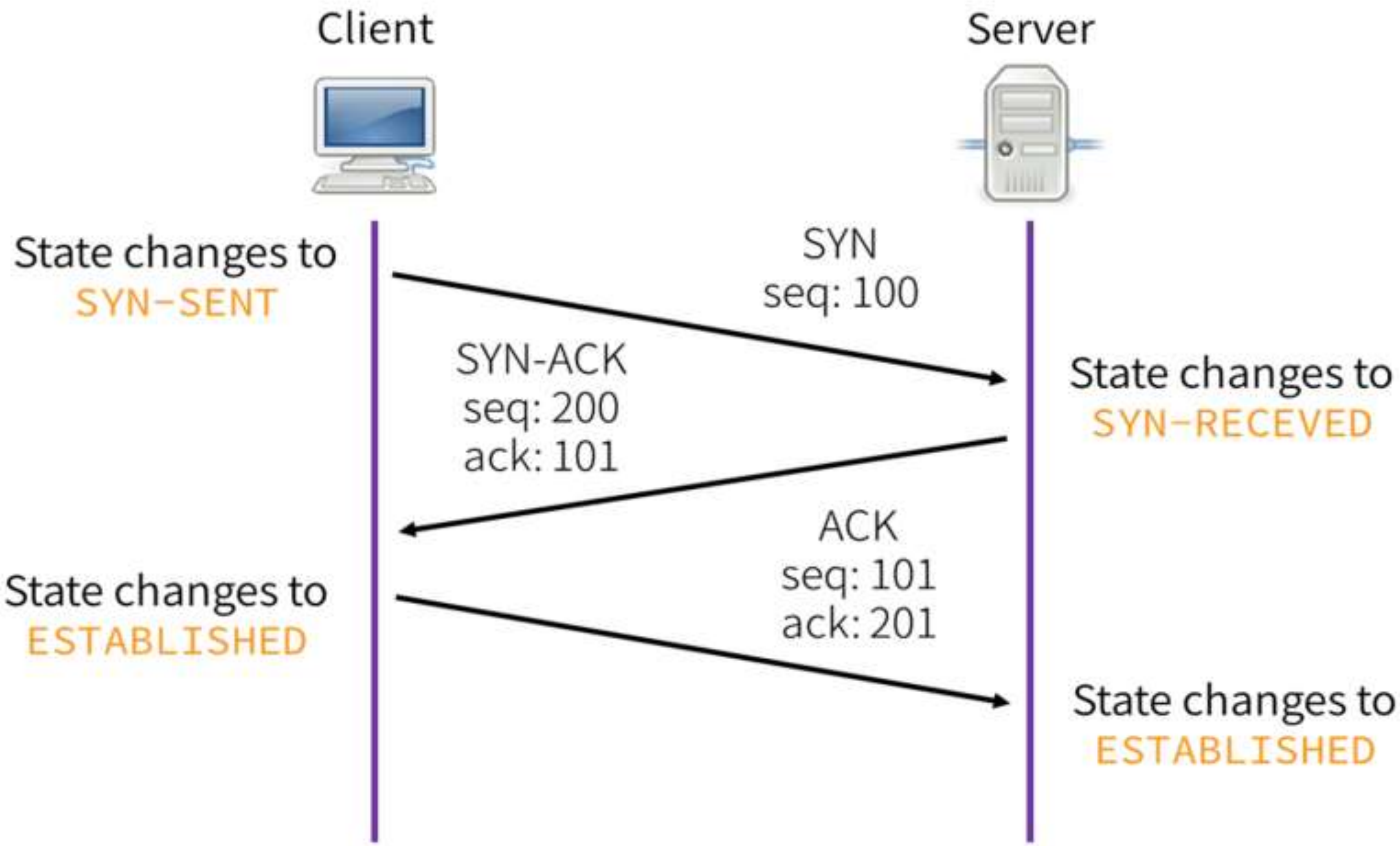
- Reliable, in-order delivery (TCP)
  - congestion control
  - flow control
  - connection setup
- Unreliable, unordered delivery: UDP
  - no-frills extension of “best-effort” IP
- services not available:
  - delay guarantees
  - bandwidth guarantees



TCP Transport Protocol



TCP Connection Establishment



Socket programming

- **Goal**:
  - Learn how to build client/server application that communicate using sockets
  - Learn what happens in your server/client programs under the hood

socket

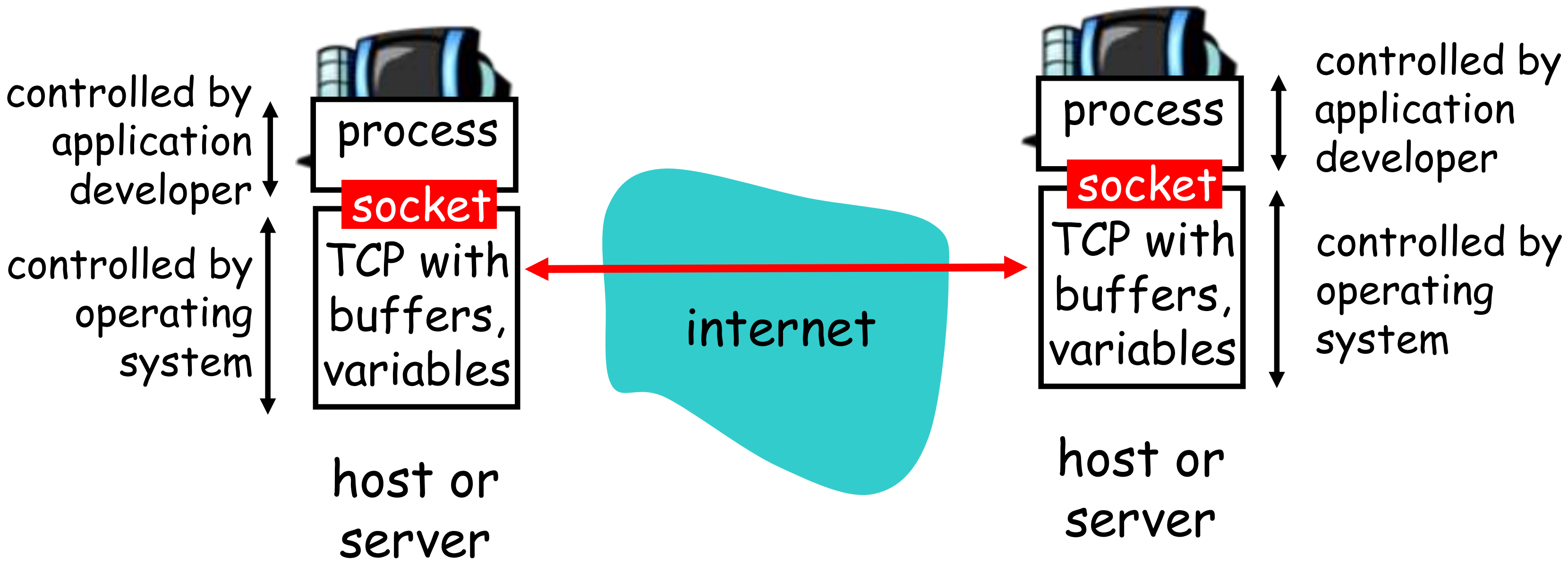
a *host-local*, *application-created*, *OS-controlled* interface (a "door") into which application process can both send and receive messages to/from another application process

Socket API

- introduced in BSD4.1 UNIX, 1981
- explicitly created, used, released by apps
- client/server paradigm
- two types of transport service via socket API:
  - unreliable datagram
  - reliable, byte stream-oriented

Socket-programming using TCP

- **Socket**: a door between application process and end-end-transport protocol (UCP or TCP)
- **TCP service**: reliable transfer of bytes from one process to another



Socket programming: Terminology

Term	Format	Example	Analogy-phone
IP(v4)	x.x.x.x	192.168.0.1	Phone number
Port	N<65535	80	Extension #
Socket	[memory/resource]	s=socket(…)	Physical phone

Ports, Services, and Hosts

- **Services** running on a **host** bind to a **port**, and use that port as a means of data transmission, which looks like slightly fancy file writing in Linux
- Ports allow us to run many services off the same IP, so 192.168.1.5:80 would handle all the HTTP traffic for the host at 192.168.1.5
- Well known (and permissions regulated) ports are 0 < x <= 1024
- The rest of the port range (1025 < x <= 65535) is open and available
- Here are a few well known ports you should be familiar with

80	443	22	25	53	389	67
HTTP	SSL/TLS	SSH/SFTP	SMTP	DNS	LDAP	DHCP

Client/server socket interaction: TCP

Server (stand-by, waiting for requests)	Client (initiate the request)
Create socket (Claim resources/ available phone)	Create socket (Claim resources/ available phone)
Bind port (Claim ID on this machine/ get a phone extension No.)	- (Don’t care about source port/ phone #)
Listen & accept (Listen: Wait for connections/ Wait for phone call Accept: Accept connection/ Answer phone call)	Connect (connect to server IP:PORT/ call phone #))
Send/ receive (Communication/ Chat on phone)	Send/ receive (Communication/ Chat on phone)
Close socket (End communication/ Hang up the phone)	Close socket (End communication/ Hang up the phone)

Red words: wait for the other side

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

Client/server socket interaction: TCP

Server (stand-by, waiting for requests)

Create socket
(Claim resources/ available phone)

Bind port
(Claim ID on this machine/
get a phone extension No.)

Listen/ accept

Fork/ New thread

Parent

Child

Send/ receive
(Communication/ Chat on phone)

Close socket
(End communication/ Hang up the phone)

Client (initiate the request)

Create socket
(Claim resources/ available phone)

Connect
(connect to server/ call others)

Send/ receive
(Communication/ Chat on phone)

Close socket
(End communication/ Hang up the phone)

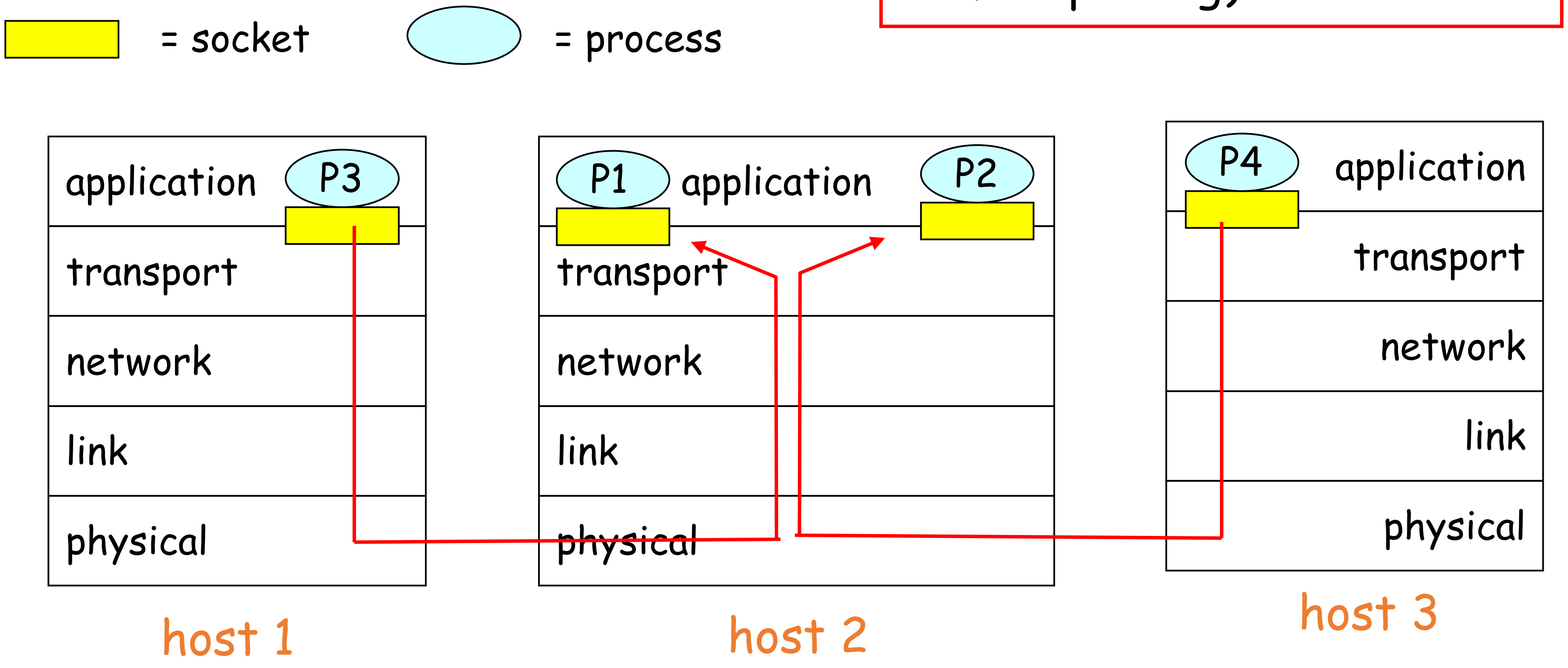
Multiplexing/Demultiplexing

Demultiplexing at rcv host:

delivering received segments to correct socket

Multiplexing at send host:

gathering data from multiple sockets, enveloping data with header (later used for demultiplexing)



Connection-oriented demux

- TCP socket identified by 4-tuple:
  - source IP address
  - source port number
  - dest IP address
  - dest port number
- recv host uses all four values to direct segment to appropriate socket
- Server host may support many simultaneous TCP sockets:
  - each socket identified by its own 4-tuple
- Web servers have different sockets for each connecting client
  - There are exceptions though

Socket programming Example: Server side

Create socket

(Claim resources/ available phone)

Bind port

(Claim ID on this machine/
get a phone extension No.)

Listen/ accept

Send/ receive

(Communication/ Chat on phone)

Close socket

(End communication/ Hang up the phone)

```python
#!/usr/bin/env python2
# Echo server program
import socket
import sys

HOST = ""                 # Symbolic name meaning all available interfaces
PORT = 50007               # Arbitrary non-privileged port
try:
    s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
except OSError as msg:
    s = None
try:
    s.bind((HOST, PORT))
    s.listen(128)
except OSError as msg:
    s.close()
    s = None
if s is None:
    print('could not open socket')
    sys.exit(1)
try:
    conn, addr = s.accept()
    print('Connected by', addr)
    while True:
        data = conn.recv(1024)
        if not data:
            break
        conn.send(data)
except:
    print("Exception")
```

Testing the example

- Allow connections to the port
  - Depends on the previous configurations (the chains etc.)
  - Fast solution (Dangerous, don’t do this in production environments):
    - sudo iptables -F
- Run the script:
  - python server.py
- Check the port is listened:
  - sudo ss -tulpn | grep LISTEN
- Connect to the port:
  - nc [IP] 50007

Socket programming Example: Client side

```python
#!/usr/bin/env python2
#Echo client program
import socket
import sys

HOST = '192.168.1.14'    # The remote host
PORT = 50007              # The same port as used by the server
s = None
try:
    s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
except OSError as msg:
    s = None
try:
    s.connect((HOST, PORT))
except OSError as msg:
    s.close()
    s = None
if s is None:
    print('could not open socket')
    sys.exit(1)
try:
    s.sendall(b'Hello, world')
    data = s.recv(1024)
    print('Received', repr(data))
except:
    s.close()
    print("Exception")
```

Create socket
(Claim resources/ available phone)

Connect
(connect to server/ call others)

Send/ receive
(Communication/ Chat on phone)

Close socket
(End communication/ Hang up the phone)

Traffic Analysis

- The communicated traffic can be captured using tcpdump:
  - sudo tcpdump tcp port 50007 -w capture_50007.pcap