Client-Server Model and Socket Programming

Client-Server Model

- Many network applications would involve getting services from a remote machine
  - e.g. web, cloud
- These applications have two pieces: client and server
- Client sends requests, e.g. browser sends the files the user wants to the web server
- Server provides service, e.g. web server sends the content requested to the clients

Process Communication Across Network

- Recall that each program in execution is a process
  - Web browser is a process, web server is also a process
- Both the client and server machines would be hosting multiple processes at the same time
- When the client process sends a request, it has to be properly addressed to the correct process, not only the machine itself!

Sender and destination information

Address of user machine: IP address

To identify the process : port number is also needed

Each IP packet carries:
Destination IP
Destination port number
Source IP
Source port number

Destination IP address allows the network routers to deliver the packet to the right machine

Destination port number allows the machine to deliver the packet payload to the right process

More about Port Numbers

- Each port number is a 16-bit number
  - Range of port numbers: 0 – 65535
- 0 – 1023 are reserved for well-known applications
  - e.g. the port number for webserver is 80
- Numbers from 1024 – 65535 can be used by other applications

Socket Programming

- Socket is a facility for a machine to establish a process-to-process connection with another machine
- Sockets allow application programmers to use the standard mechanisms in network hardware and operation system to build their network applications
- Stream sockets: connection-oriented (TCP)
- Datagram sockets: connection-less (UDP)

Multiplexing/demultiplexing

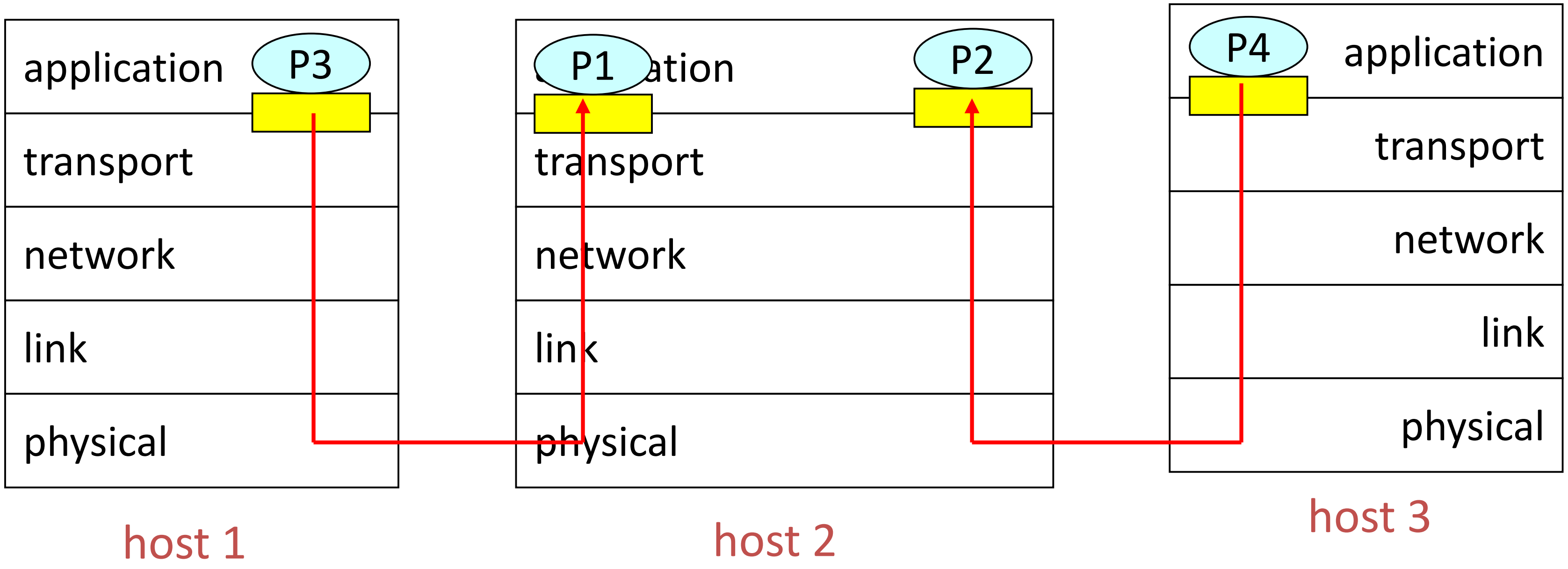
Demultiplexing at rcv host:

delivering received segments to correct socket

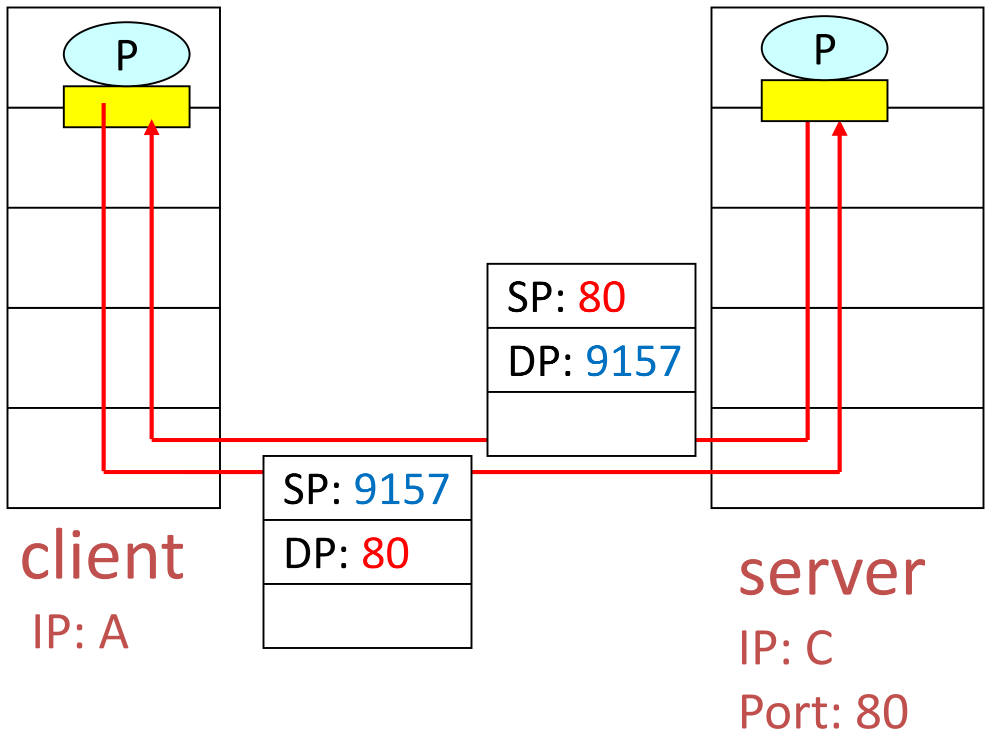
Multiplexing at send host:

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

🟨 = socket ⬭ = process



TCP Connections



Step 1: Server listens at a port that is known to the clients (say 80) and waits for connection

Step 2: Client sends request to server with port number 80. Client also picks a port number as source port for identifying its local process (9157 in the example)

Step 3: When server receives the request, it binds the socket, and sends reply to the client with the port information carried in the request.

Client-Server Simple Example

- Files: `echo_server.py` & `echo_client.py`
- A simple echo server
  - server sends back what the client sends and then closes the connection
  - both server and client processes are in the same machine
- Execution of the program
  - start the server first on one terminal
  - start the client on the other
  - another client can start again without launching another server

Python Server Implementation

- Import the `socket` library
  - `import socket`
- Create socket
  - `sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)`
- Select port number for listening
  - `server_address = ('', 12345)`
- Bind socket and listen to the channel
  - `sock.bind(server_address)`
  - `sock.listen(1)`
- Accept incoming connection
  - `connection, client_address = sock.accept()`
- close the connection
  - `connection.close()`

Python Client Implementation

- Import the `socket` library and create socket as in server
- Specify server address and server port
  - `server_address = ('<server IP>', 12345)`
- Attempt to connect to server
  - `sock.connect(server_address)`
- close the connection
  - `sock.close()`

Data Exchange (1)

- data exchange through "utf-8" encoded string
  - data should be encoded before sending out
  - received data may need to be decoded before printing out
- Sending a string
  - Use the `send()` function "utf-8" encoding
  - e.g. `send(message.encode("utf-8"))`
- receiving data
  - `recv(n)` function where `n` is the length of data being received

Data Exchange (2)

- server sends and receives data through the connection (refer to `echo_server.py`)

```
data = connection.recv(1024)
print('received "%s"' % data.decode("utf-8"))
connection.send(data)
```

- client sends and receives data through the socket (refer to `echo_client.py`)

```
sock.send(message.encode("utf-8"))
data = sock.recv(1024)
print('received "%s"' % data.decode("utf-8"))
```