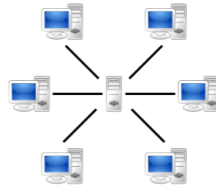


Application Layer (I)

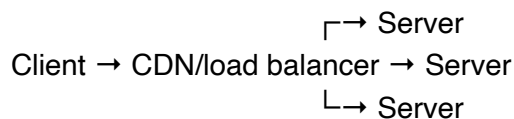
Two types of architecture

- Client and Server



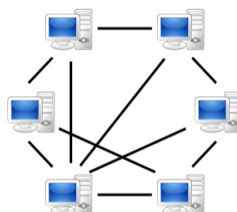
Img ref: <https://en.wikipedia.org/wiki/Peer-to-peer#/media/File:Server-based-network.svg>

- **Client** (caller)
 - actively opens connection to the server
 - resource requesters
 - must know server's IP address and port number
- **Server** (callee)
 - waits for clients to connect
 - resource providers
 - may handle multiple simultaneous client connections
- Examples:
 - web browser <-> web server
 - email client <-> mail server
 - SSH client <-> SSH server
- “Server” doesn’t necessarily mean one physical machine. When your browser connects to a service such as Google, Netflix, or ChatGPT, the Internet service may involve:



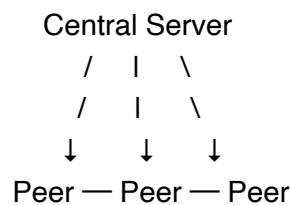
- So, client/server describes the communication relationship, not necessarily the physical deployment. A server is fundamentally a role played by a process, not a particular kind of computer.

- P2P (peer-to-peer)



Img ref: <https://en.wikipedia.org/wiki/Peer-to-peer#/media/File:P2P-network.svg>

- Peer
 - **not owned by the service provider**, but are instead desktops and laptops controlled by users
 - **equal privileged**
 - both client and server, client when requesting resources from other peers, server when providing resource to other peers
- **Minimal (or no) reliance on dedicated servers** in data centers, as peers may join and leave dynamically.
- Application exploits **direct communication between pairs** of intermittently connected hosts/peers
 - Application examples:
 - file sharing (BitTorrent)
 - Blockchain
- Some applications have **hybrid architectures**
 - For example:



- **Servers** are used to **track the IP addresses of users**, providing authentication, peer discovery, metadata, and coordination, But user-to-user messages are sent directly between user hosts.
- Advantages:
 - **Self-scalability**
 - E.g., in a P2P file-sharing application, although each **peer generates workload** by requesting files, each peer **also adds service capacity** to the system by distributing files to other peers.
 - **Cost effective**
 - Normally **don't require significant server infrastructure** and server **bandwidth** (in contrast with clients-server designs with datacenter)
- Challenges:
 - **ISP Friendly**
 - Most residential ISPs (including DSL and cable ISPs) have been dimensioned for "asymmetrical" bandwidth usage, that is, for **much more downstream than upstream traffic**. But P2P video streaming and file distribution applications **shift upstream traffic from servers to residential ISPs**, thereby putting significant stress on the ISPs. Future P2P applications need to be designed so that they are friendly to ISPs.
 - **Security**
 - Due to the highly distributed and open nature

- Incentives

Underlying Point-to-point Communication

- Underlying network provides **reliable connection-oriented** or **unreliable connection-less** service (TCP or UDP)
- A quadruple to uniquely identify an application communication between a client and a server
 - **source/dest IPs, source/dest port numbers** (port #, id that is unique within the host)
 - Port Numbers:
 - 16-bit numbers, $0 \leq \text{port} \leq 65535$
 - Some familiar service ports: HTTP (80), HTTPS (443), SSH (22), SMTP (25), DNS (53)
 - Client typically uses *ephemeral* source (local) port number, assigned by the OS
 - Server connects to local port from typically a *well-known* one
 - Why does it need a quadruple to uniquely identify a communication?
 - Both client and sever can have **multiple processes running simultaneously**. A client can send multiple web requests to different websites, and a server can receive multiple requests from different clients.
 - E.g., open a web browser on your laptop and then try to connect to CS331 and CS333 course websites. Both course websites are located on the Colby CS server. Destination port 80 can let server know that the two requests are web requests, the source port can let the server tell which response is to which request.

Command Line Utility: nc

- Netcat (nc), an utility for reading from and writing to network connections using TCP or UDP.
- It can be driven by other programs or scripts.
- Good tool for network debugging and investigation.
 - E.g., Scan all ports up to 100

```
d122-01:~> nc -z -v d122-05.colby.edu 1-100
```

- REF
 - <https://en.wikipedia.org/wiki/Netcat>
 - <https://www.digitalocean.com/community/tutorials/how-to-use-netcat-to-establish-and-test-tcp-and-udp-connections-on-a-vps>

- Another application of using nc is to transfer files

Client

```
mbp-190250:~ yingli$ echo "Hello gloin" > test.txt
mbp-190250:~ yingli$ cat test.txt
Hello gloin
mbp-190250:~ yingli$ nc gloin.cs.colby.edu 12345 < test.txt
mbp-190250:~ yingli$
```

Server

```
gloin:~> nc -l 12345 > output.txt
gloin:~> cat output.txt
Hello gloin
gloin:~>
```

- In the above example, I was trying to send a file, test.txt, from my laptop to gloin, and saved the file into output.txt.
- The gloin is the server in this case, waiting for the file from client, my laptop, listening on port 12345.
- My laptop is the client in this case, which connects to the server on port 12345 and sends files to it.

- E.g., SSH to dori and gloin for the example.

```

yingli — ssh yingli
[dori:~> nc gloin.cs.colby.edu 12345
Hi gloin, this is dori
Hi dori, I'm gloin, nice to meet you

```

Client

2. connecting the client to a server on gloin on port 12345
3. typing a message to the server "Hi gloin, this is dori"

```

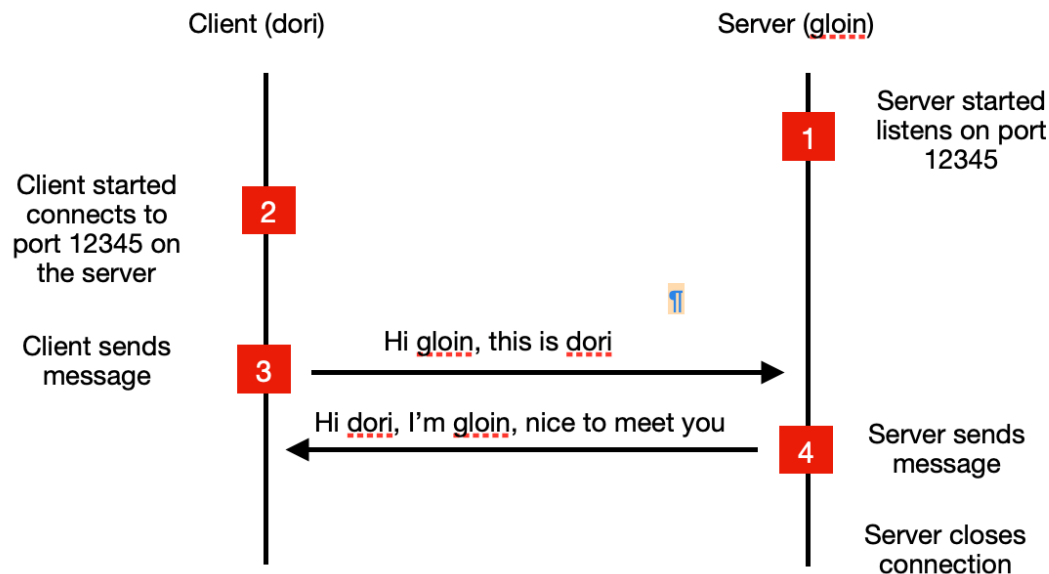
yingli — ssh yingli@gloin
gloin:~> nc -l 12345
Hi gloin, this is dori
Hi dori, I'm gloin, nice to meet you

```

Server

1. starting server on port 12345
4. typing response to the client and breaking the connection by type CTRL + C

- Sequence diagram for the above example



Socket API

- We can use nc on the command line to build a connection between two nodes. We can also use socket API to set up a connection when programming.
 - Socket API is designed to support any protocol - not just TCP/UDP/IP.
 - It was originally defined in C, but adopted by essentially all programming languages.
-

Perspectives

- We can use parameters in Socket API to specify a connection is a reliable, stream-oriented service (TCP) `SOCK_STREAM`, or an unreliable, datagram service (UDP) `SOCK_DGRAM`.
 - Socket API also leaves parameters to specify a connection is built on IPv4, `AF_INET` or IPv6, `AF_INET6`.
-

Python Socket Programming (TCP)

- In this class, we use Python to build a TCP connection over IPv4 as an example. But, you can also create an UDP over IPv4 or IPv6 by changing the parameters. You are also encouraged to explore Socket API in other programming languages, which will be a good exercise.
- To build a connection, we need to program a client and server.
 - The server should be launched first and wait for the connection request from the client.
 - The client connects to the server using the port the server is listening and server's IP address. In this example, the client uses 'localhost,' indicating that the server is running on the same computer, which will be translated into a local IP address automatically.

- Client side:

- To use socket API, the socket module should be imported first.

```
from socket import *
```

- To build a connection, a socket instance should be created first, which specifies the IP protocol and TCP/UDP service supporting a connection.

```
socket(AF_INET, SOCK_STREAM)
```

- `AF_INET`: IPv4; `AF_INET6`: IPv6
- `SOCK_STREAM`: TCP; `SOCK_DGRAM`: UDP
- `AF_INET + SOCK_STREAM` → IPv4 + TCP
- `AF_INET6 + SOCK_STREAM` → IPv6 + TCP
- `AF_INET + SOCK_DGRAM` → IPv4 + UDP

- Then, we can invoke the connect function of the socket instance to connect to the other point specified by the server name and port number.

```
connect((serverName, serverPort))
```

- The send function transmits the data to the other point over the socket.

```
send(sentence.encode())
```

- The recv function receives the data sent by the other point. The buffer size is specified by the parameter of this function in bytes.

```
recv(1024)
```

- When it's done, we need to mark the socket closed.

```
close()
```

- Server side:

- After creating a socket instance, the program should use bind function to assign an IP and a port to the socket instance.

```
bind(('localhost', serverPort))
```

- Then, the server should be listening to the incoming requests.

```
listen(0)
```

- When a client connect request is arrived, the server accept this connection.

```
accept()
```

- It then can receive data from the other end and send response if necessary.

```
recv(1024); send(capitalizedSentence.encode())
```

- When it's done, we need to mark the socket closed.

```
close()
```

The screenshot displays two Python scripts and their execution in terminal windows.

TCPClient.py (Left Panel):

```
1 from socket import *
2
3 serverName = 'localhost'
4 serverPort = 12000
5 modifiedSentence = None
6
7 clientSocket = socket(AF_INET, SOCK_STREAM)
8 clientSocket.connect((serverName, serverPort))
9 sentence = input('Input lowercase sentence:')
10 clientSocket.send(sentence.encode())
11 modifiedSentence = clientSocket.recv(1024)
12 print('From Server: ', modifiedSentence.decode())
13 clientSocket.close()
```

TCPServer.py (Left Panel):

```
1 from socket import *
2
3 serverPort = 12000
4 serverSocket = socket(AF_INET, SOCK_STREAM)
5 serverSocket.bind(('localhost', serverPort))
6 serverSocket.listen(0)
7 capitalizedSentence = None
8
9 print("The server is ready to receive")
10 while True:
11     connectionSocket, addr = serverSocket.accept()
12     sentence = connectionSocket.recv(1024).decode()
13     capitalizedSentence = sentence.upper()
14     connectionSocket.send(capitalizedSentence.encode())
15     connectionSocket.close()
```

Terminal Windows (Right Panel):

- Top Terminal (Running TCPClient.py):**

```
mbp-190250:SimpleVersion yingli$ python TCPClient.py
Input lowercase sentence:hello world
('From Server: ', u'HELLO WORLD')
mbp-190250:SimpleVersion yingli$
```
- Bottom Terminal (Running TCPServer.py):**

```
mbp-190250:SimpleVersion yingli$ python TCPServer.py
The server is ready to receive
```

How to modify the code so that the connection between the client and server can end till the user input "quit"? Otherwise, the client can keep sending strings to the server and receive capitalized strings (while loop before line 7 in client side).

🔌 TCPClient } No Selection	🔌 TCPServer } No Selection
<pre>1 from socket import * 2 3 serverName = 'localhost' 4 serverPort = 12000 5 modifiedSentence = None 6 7 while modifiedSentence != 'QUIT': 8 clientSocket = socket(AF_INET, SOCK_STREAM) 9 clientSocket.connect((serverName, serverPort)) 10 sentence = input('Input lowercase sentence:') 11 clientSocket.send(sentence.encode()) 12 modifiedSentence = clientSocket.recv(54321).decode() 13 print('From Server:', modifiedSentence) 14 clientSocket.close()</pre>	<pre>1 from socket import * 2 3 serverPort = 12000 4 serverSocket = socket(AF_INET, SOCK_STREAM) 5 serverSocket.bind(('', serverPort)) 6 serverSocket.listen(0) 7 capitalizedSentence = None 8 9 print("The server is ready to receive") 10 while capitalizedSentence != "QUIT": 11 connectionSocket, addr = serverSocket.accept() 12 sentence = connectionSocket.recv(54321).decode() 13 capitalizedSentence = sentence.upper() 14 connectionSocket.send(capitalizedSentence.encode()) 15 connectionSocket.close()</pre>

Disclaimer: Notes adapted from the textbook and online resources.