Chapter 26 Fundamentals of Computer Networks (Part 2)

Content

Syllabus Learning Outcomes

4.1 **Fundamentals of Computer Networks**

Understand computer network technology

4.1.5 Explain client-server architecture.

4.1.6 Implement an iterative server with socket programming. Given the server code, students should be able to implement the client code for a given scenario, and vice-versa, e.g. for a tic-tac-toe game.

1 Introduction

When we use the internet, we often use a web browser such as Internet Explorer, Firefox, or Chrome. The web browser is an example of an application, meaning a program used by the end user. It is also referred to as a client because it starts the exchange of information. It requests web pages from a web server, which is a different program running on another device. A server serves the information requested by the client. This is the client-server model, and most programs that access the internet follow it.

Applications such as email, file transfer, media streaming, and online games all communicate via the internet with their respective servers.

In peer-to-peer networking, each device can act as a client requesting files from another device, as a server sharing files with another device, or as both simultaneously.

Messages sent between the application (i.e. the client) and the server use a specific protocol designed for the application. The application layer provides these protocols.

2 Computer Network Architecture: Client and Server Processes

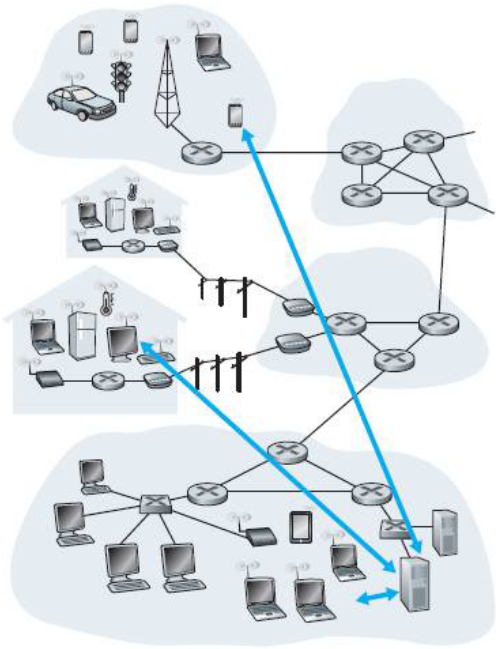
A network application consists of pairs of processes that send messages to each other over a network. For each pair of communicating processes, we typically label one of the two processes as the client and the other process as the server.

In a Web application, a browser is a client process, and a Web server is a server process. A browser process initialises contact with a Web server process. Hence, the browser process is the client, and the Web server process is the server.

In a Peer-to-peer (P2P) file-sharing system, a file is transferred from a process in one peer to a process in another peer. The peer that is downloading the file is labelled as the client, and the peer that is uploading the file is labelled as the server. A process can be both a client and a server. For example, a process in a P2P file-sharing system can both upload and download files. When Peer A asks Peer B to send a specific file, Peer A is the client and Peer B is the server in the context of this specific communication session. Sometimes, we also use the terminology “client side and server side of an application.”

In the context of a communication session between a pair of processes, the process that initiates the communication is labelled as the client. The process that waits to be contacted to begin the session is the server.

2.1 Client-server Architecture



Source: Kurose, J., & Ross, K. (2016). Computer Networking: A Top-Down Approach (7th ed.). Pearson.

In a client-server architecture, there is an always-on host, called the server, which services requests from many other hosts, called clients. A classic example is the Web application for which an always-on Web server services requests from browsers running on client hosts. When a Web server receives a request for an object from a client host, it responds by sending the requested object to the client host. The clients, i.e. the web browsers in this case, do not directly communicate with each other.

Another characteristic of the client-server architecture is that the server has a fixed, well-known address called an IP address. Because the server has a fixed, well-known address, and because the server is always on, a client can always contact the server by sending a packet to the server's IP address. Some of the better-known applications with a client-server architecture include the Web, FTP, Telnet and e-mail.

Often in a client-server application, a single-server host is incapable of keeping up with all the requests from clients. For this reason, a data centre, housing a large number of hosts, is often used to create a powerful virtual server. The most popular Internet services such as search engines (e.g. Google, Bing, Baidu), Internet commerce (e.g. Amazon, eBay, Alibaba), Web-based email (e.g. Gmail and Yahoo Mail), social networking (e.g. Facebook, Instagram, Twitter, and WeChat) employ one or more data centres. Google, for example, has 30 to 50 data centres distributed around the world, which collectively handle search, YouTube, Gmail, and other services. A data centre can have hundreds of thousands of servers, which must be powered and maintained.

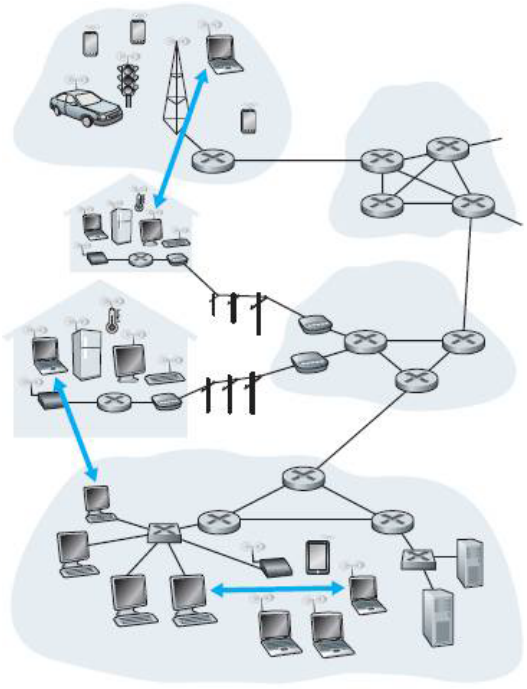
Additionally, the service providers must pay recurring interconnection and bandwidth costs for sending data from their data centres.

The advantages and disadvantages of a client-server network can be summarised as follows:

Some examples where a client-server network may be used:
1. Large enterprises
2. Emails
3. File Transfer Protocol (FTP)
4. Online banking systems

Advantages	Disadvantages
Centralised control of data and resources	Higher initial cost due to the need for specialised high-performance servers
Easy to schedule backups of all shared files at regular intervals	Administrative costs needed for the maintenance of servers and clients
Security may be enhanced with the use of specialised software or operating system features that are designed for servers	

2.2 Peer-to-peer (P2P) Architecture



Source: Kurose, J., & Ross, K. (2016). Computer Networking: A Top-Down Approach (7th ed.). Pearson.

In a P2P architecture, there is minimal or no reliance on dedicated servers in data centres. Instead, the application exploits direct communication between pairs of intermittently connected hosts, called peers. The peers are not owned by the service provider, but are desktops and laptops controlled by users, residing in homes, universities, and offices. Because the peers communicate without passing through a dedicated server, the architecture is called peer-to-peer.

Many of today's most popular and traffic-intensive applications are based on P2P architectures. These applications include file sharing (e.g. BitTorrent), peer-assisted download acceleration (e.g. Xunlei), and Internet telephony and video conference (e.g. Skype).

The advantages and disadvantages of a Peer-to-Peer architecture can be summarised as follows:

Some examples where a peer-to-peer network may be used:

1. Homes and small businesses
2. Unrestricted sharing of files and data
3. Torrent downloads
4. Television and broadcasting

Advantages	Disadvantages
Cheaper to set up as there is no cost related to dedicated servers; basic computers can act as servers to share resources	More effort is required to access back up resources as they are stored locally within each computer instead of centrally in a server
Easy to set up as no specialised software or operating system features are needed	Security is low as access rights are handled by individual computers; not administered by a central server
Storage of data is decentralised and can be carried out by individual users at each computer	

2.3 Hybrid Architecture

Some applications have hybrid architectures, combining both client-server and P2P elements. For example, for many instant messaging applications, servers are used to track the IP addresses of users, but user-to-user messages are sent directly between user hosts, without passing through intermediate servers.

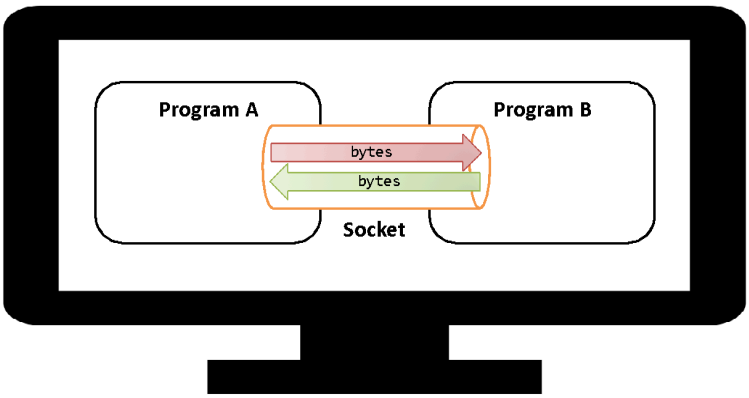
One of the most compelling features of P2P architectures is their self-scalability. For example, in a P2P file-sharing application, although each peer generates workload by requesting files, each peer also adds service capacity to the system by distributing files to other peers.

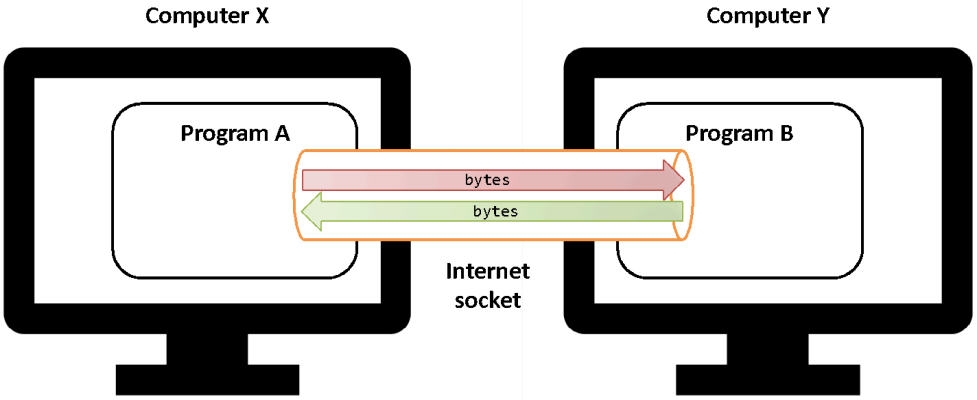
P2P architectures are also cost-effective, since they normally don't require significant server infrastructure and server bandwidth in contrast with client-server designs with data centres. However, P2P applications face challenges of security, performance, and reliability due to their highly decentralised structure.

3 The Interface Between the Process and the Computer Network

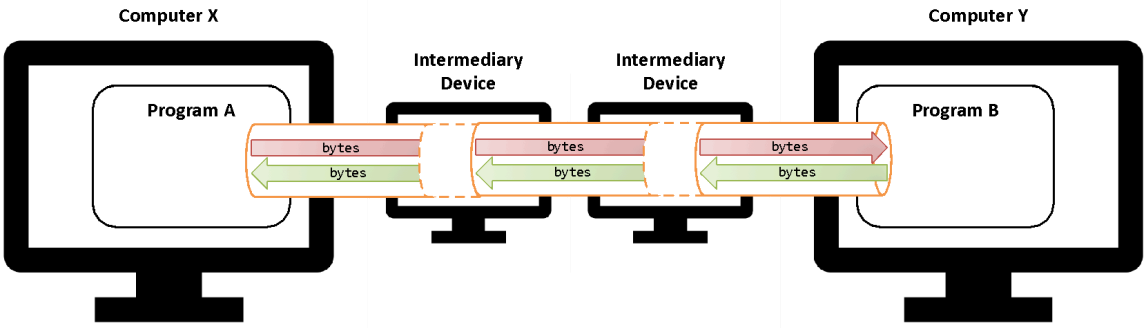
As noted above, most applications consist of pairs of communicating processes, with the two processes in each pair sending messages to each other. Any message sent from one process to another must go through the underlying network. A process sends messages into and receives messages from the network through a software interface called a socket. You can picture a socket connection as a pipe between two running programs. The pipe is bidirectional and can carry data (represented by bytes) in both directions.

There are many kinds of sockets, but the kind that is most often discussed is called an Internet socket. Internally, Internet sockets deliver data using the same Transmission Control Protocol and Internet Protocol suite (TCP/IP) that is used to transmit data over the Internet. This means that Internet sockets can deliver data between any two programs, programs that are running on different computers, as long as the two computers can access each other over the network.





For simplicity, we illustrate an Internet socket as a pipe that is only attached to the two computers. In reality, however, data that is transmitted through an Internet socket may pass through multiple devices before reaching its destination. You should be aware that any of these devices can steal or modify the data that passes through a socket unless you encrypt the data first. A more accurate illustration of a socket that shows how the data passes through multiple devices is shown below:

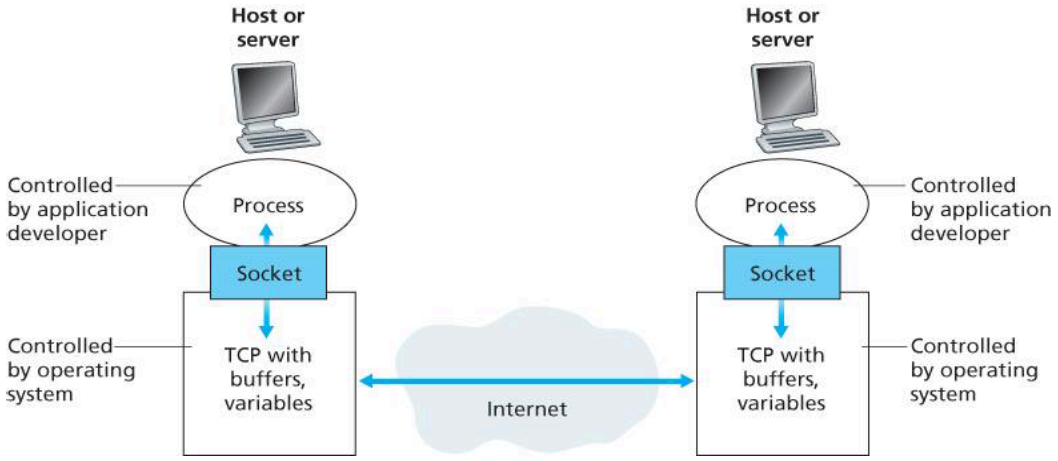


3.1 Socket Communication

The figure below illustrates socket communication between two processes that communicate over the Internet, with the TCP protocol as the underlying transport protocol.

Why do we need a protocol? As networks can become congested, we cannot assume that data sent over Internet sockets will be transmitted instantaneously. For instance, a program may receive only the first half of a message before the second half arrives some time later. To avoid working with incomplete data, we will need to define a protocol so that the start and end of messages can be detected unambiguously.

As shown in the figure, a socket is the interface between the application layer and the transport layer within a host. It is also referred to as the Application Programming Interface (API) between the application and the network, since the socket is the programming interface with which network applications are built. The application developer has control of everything on the application layer side of the socket but has little control of the transport-layer side of the socket.



Source: Kurose, J., & Ross, K. (2016). Computer Networking: A Top-Down Approach (7th ed.). Pearson.

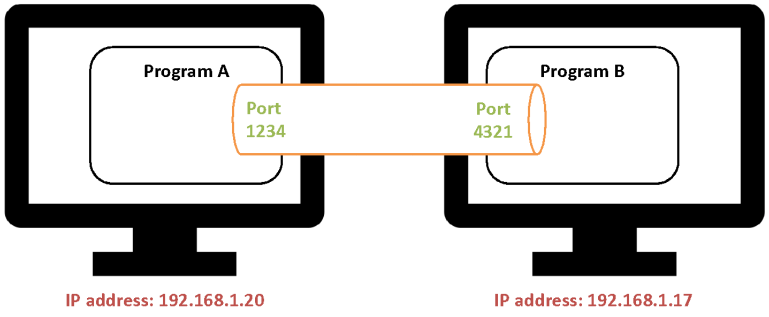
The only control that the application developer has on the transport-layer side is:

- the choice of transport protocol, if such provision is available and,
- the ability to fix a few transport-layer parameters such as maximum buffer and maximum segment sizes.

3.2 IP addresses and Ports

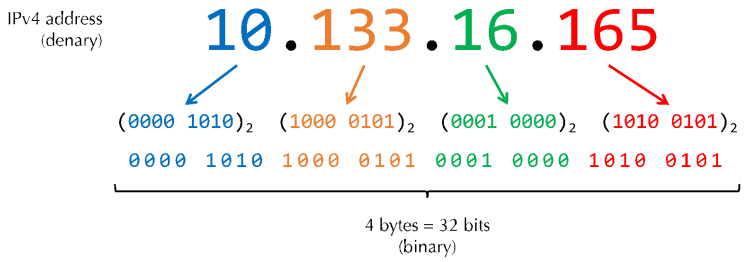
In order for a process running on one host to send packets to a process running on another host, the receiving process needs to have an address. To identify the receiving process, two pieces of information need to be specified:

- The IP address of the host and,
- an identifier that specifies the receiving process in the destination host, i.e. port number.



Each end of a socket is associated with a running program and is uniquely identified by a combined **IP address** and **port number**. The IP address identifies which device that end of the socket is attached to and the port number identifies which program on that device is using the socket.

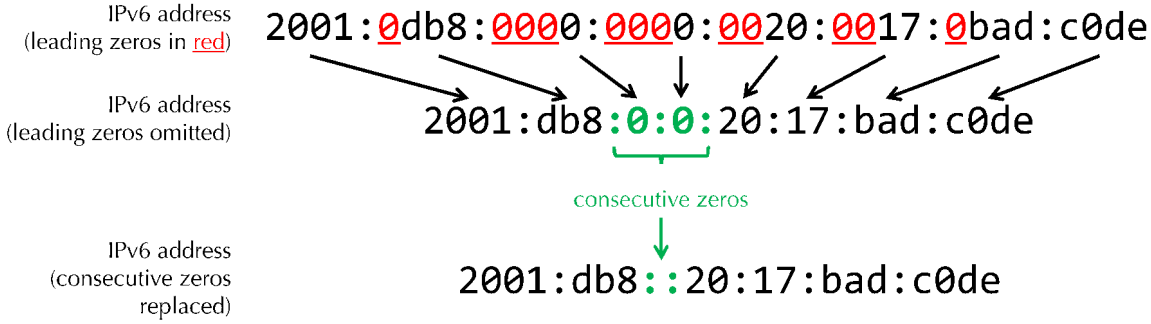
As mentioned in Fundamentals of Computer Networks Part 1, there are two kinds of IP addresses in use today: **IPv4** addresses and **IPv6** addresses. IPv4 addresses have 32 bits and are usually presented as 4 denary numbers separated by dots. Each denary number can range from 0 to 255 (inclusive) and corresponds to one byte (8 bits) of the IP address.



Some IPv4 addresses are reserved for special use and have specific meanings. Two important special IPv4 addresses are:

- 127.0.0.1 Refers to the local computer
- 0.0.0.0 Refers to all IP addresses for local computer

IPv6 addresses, on the other hand, have 128 bits and are usually presented as 8 groups of 4 hexadecimal digits separated by colons. For compactness, leading zeros and up to one consecutive sequence of zero-only groups may be omitted.



Currently, IPv4 addresses are more frequently encountered than IPv6 addresses, so to simplify our discussion, we will be working with IPv4 addresses only.

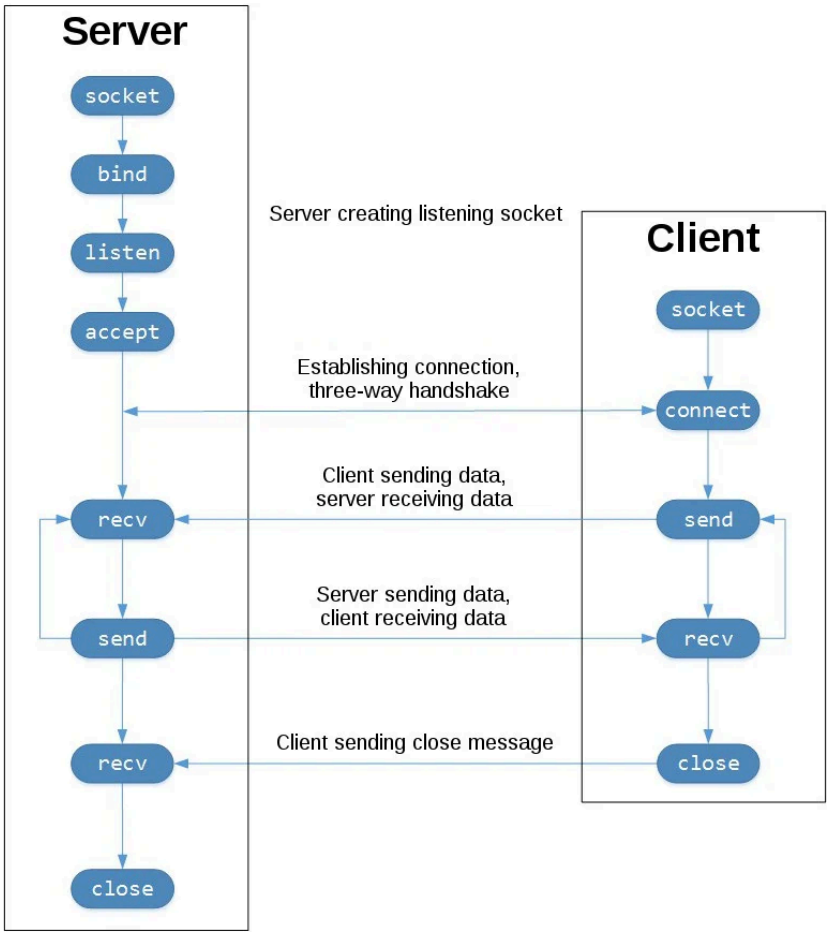
The IP address uniquely identifies the host. The sending process must also identify the receiving process, i.e., the receiving socket running in the host. This information is needed because a host could be running many network applications. A destination port number serves this purpose. On each device, port numbers are used to distinguish between attached sockets. The device also keeps track of which program is associated with each port and which port numbers are still available for use by new sockets.

Port numbers can range from 0 to 65,535. However, the first 1,024 port numbers are reserved for specific kinds of programs and should not be used for other purposes. For example, a Web server is identified by port number 80. A mail server process using the SMTP protocol is identified by port number 25. A list of well-known port numbers for all Internet standard protocols can be found at www.iana.org.

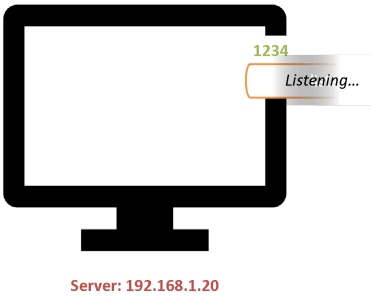
3.3 TCP Socket API

Creating a socket connection is a multi-step process that requires one program to be the server and another program to be the client. The server's IP address and port number for accepting connections must also be known ahead of time by the client.

The figure below shows an overview of the TCP Socket API.

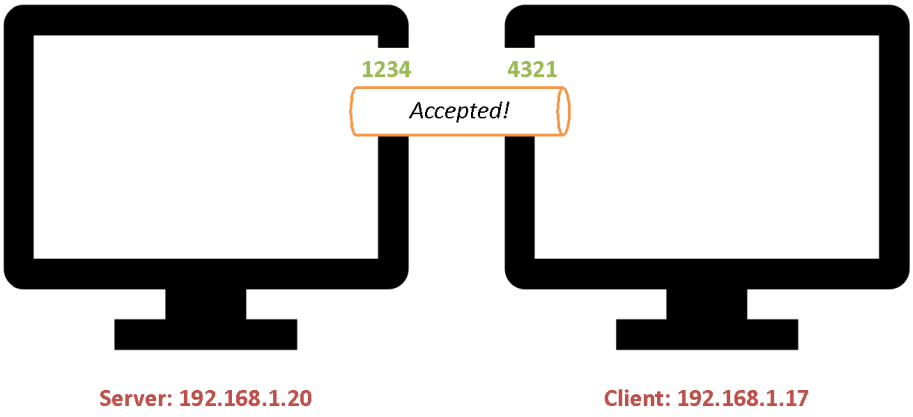


First, the server creates a passive socket, binds it to the pre-chosen port number and listens for an incoming connection. (A passive socket is not connected and merely waits for an incoming connection.)

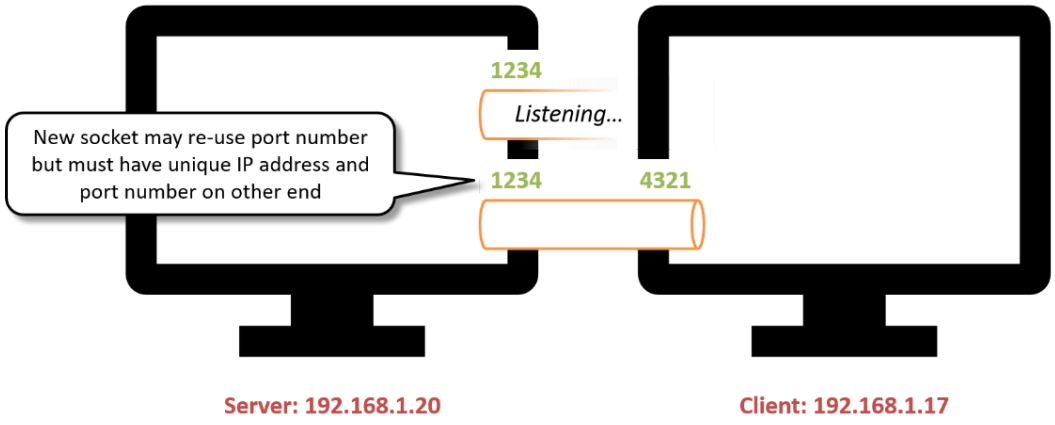


Next, the client initiates a connection request using the server's IP address and port number. If no server is listening on the chosen port, the connection will be refused.

On the other hand, if the connection request reaches an IP address and port number that a server is listening on, the server accepts and creates a new socket for the requesting client using a dynamically assigned port number.



The passive socket goes back to listening for new connections while the client and server can now exchange data using the newly-created socket.



Note that the newly-created socket is symmetrical: data sent on one end is received on the other end and vice versa. Once a socket is established, it can send data both from the client to the server and from the server to the client.

4 Socket Programing

4.1 Python's Socket Module

You can create and manage sockets in Python by importing the socket module and creating socket objects. The methods of the socket class are summarised below:

Methods	Description
`bind((host, port))`	Binds socket object to the given address tuple (host, port), where host is an IPv4 address and port is a port number
`listen()`	Enables socket to listen for incoming connections from clients
`accept()`	Waits for an incoming connection and returns a tuple containing a new socket object for the connection and an address tuple (host, port), where host is the IPv4 address of the connected client and port is its port number
`connect((host, port))`	Initiates a connection to the given address tuple (host, port), where host is the IPv4 address of the server and port is its port number
`recv(max_bytes)`	Receives and returns up to the given number of bytes from the socket
`sendall(bytes)`	Sends the given bytes to the socket

4.2 Unicode and Encodings

We are almost ready to write Python code to create our own sockets. However, as sockets work at a very basic level, they can only send and receive data in the form of raw bytes. In other words, we must be able to encode the data into a sequence of 8-bit characters using Python's bytes type.

Thankfully, a Python str can be easily converted into bytes using the str.encode() method and vice versa using the bytes.decode() method.

This encoding and decoding are necessary as internally, a Python str is actually treated as a sequence of numbers called Unicode code points. There are over a million possible code points, so it is not always possible to represent each code point using just 8 bits. Instead, the Unicode standard defines an encoding called UTF-8 so code points can be represented using bytes in a space-efficient and consistent manner.

To enter a sequence of bytes directly in code, we can use a bytes literal that starts with the letter b, followed by a sequence of bytes (in the form of ASCII characters) enclosed in matching single

or double quotes. Note that most escape codes that work for str literals also work for bytes literals.

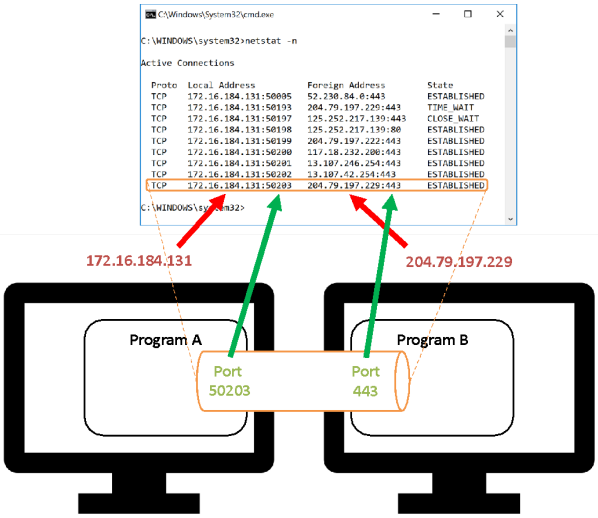
```
b'Raw bytes'

b'Raw bytes'.decode()

'Unicode str'

'Unicode str'.encode()
```

Converts bytes to str using UTF-8 encoding

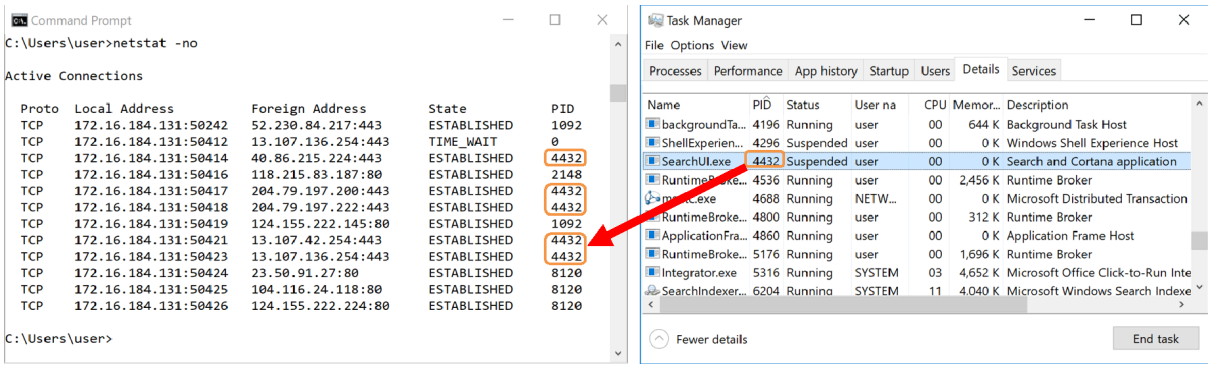
Converts str to bytes using UTF-8 encoding

4.3 Sockets

On a Windows computer with access to PowerShell or Command Prompt, you can run the command **netstat -n** to list out the sockets that are currently open on your computer. Each socket will be displayed with a combined IP address and port number for each of its ends (i.e., its local and foreign addresses):



To reveal which program is using each socket, you can run **netstat -no** to reveal the process ID (PID) associated with each socket. You can then open Task Manager and match each PID to the name of a running program:



4.4 How to Create a Basic Client and Server

Create the following basic server program that listens for a client on port 12345, accepts a connection request, sends **`b'Hello from server\n'`** to the client through the socket, then closes the socket.

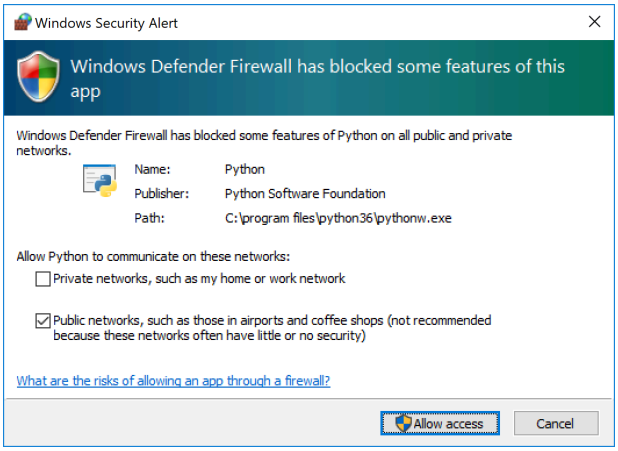
Program 1: basic_server.py

```
1   import socket
2
3   my_socket = socket.socket()
4   my_socket.bind(('127.0.0.1', 12345))
5   my_socket.listen()
6
7   new_socket, addr = my_socket.accept()
8   print('Connected to: ' + str(addr))
9   new_socket.sendall(b'Hello from server\n')
10  new_socket.close()
11  my_socket.close()
```

Note that instead of 12345 on line 4, we could have chosen any large port number to use. This number must be decided ahead of time, however, for the client to use when connecting.

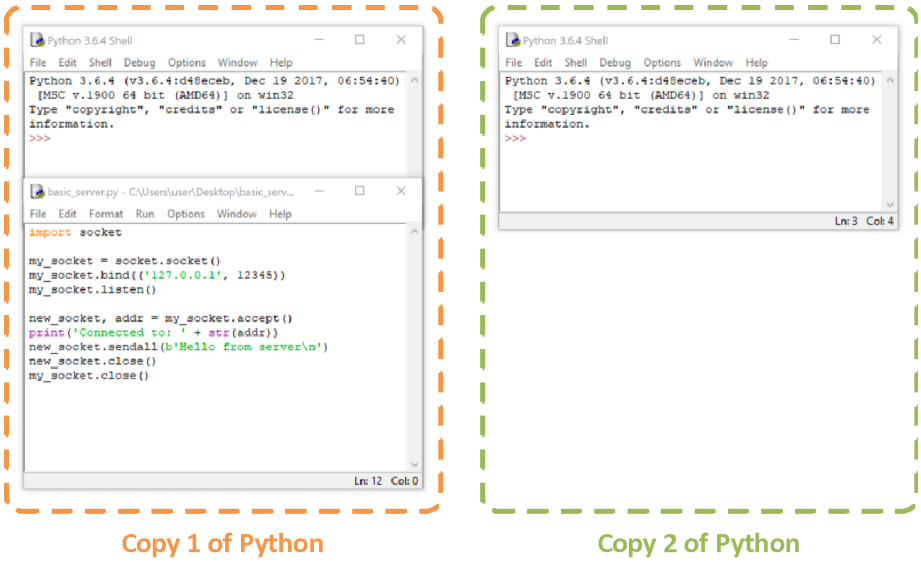
Also note that on line 7, **socket.accept()** returns a tuple of the newly created socket and a nested address tuple. We store both the new socket and the address tuple in two variables named **new_socket** and **addr** respectively. Note that **new_socket** is the socket that we actually use to send and receive data.

Run this program. If a firewall is running and has not been configured previously, you may be asked to grant Python network access at this point. Click **"Allow access"** if you are an administrator and wish to accept connection requests from other computers. Otherwise, click **"Cancel"**.



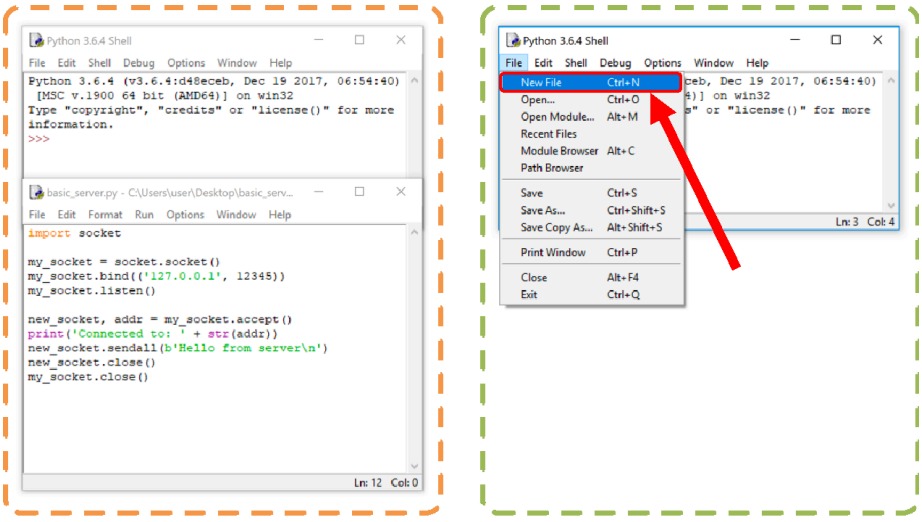
If everything is working correctly, the server should appear stuck shortly after it is started. This is because the **socket.accept()** method is blocking[1] the program and prevents it from continuing until a connection request is received.

To create a client that can connect to this server, start a second copy of Python. For instance, if you use IDLE on Windows, open the Start Menu and run IDLE again. Move any windows from the first copy of Python to one side so the two copies of Python are clearly separated.



Create a new Python program using the second copy of Python. If you use IDLE, select "New File" using the shell window that is not running the server.

[1] A “blocked” process means that it is waiting for some event to occur.



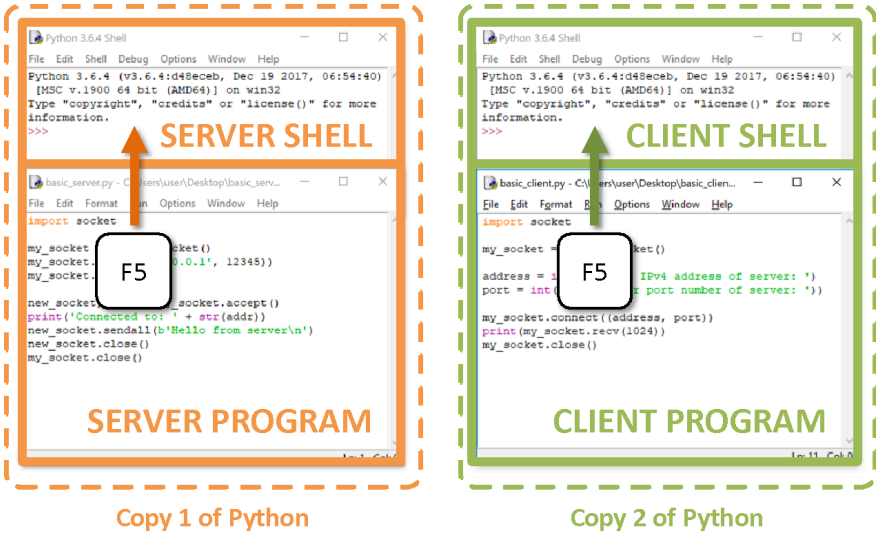
In the window that appears, enter the following basic client program that asks for the server's IP address and port number, requests for a connection, receives and prints at most 1024 bytes from the server, then closes the socket.

Program 2: basic_client.py

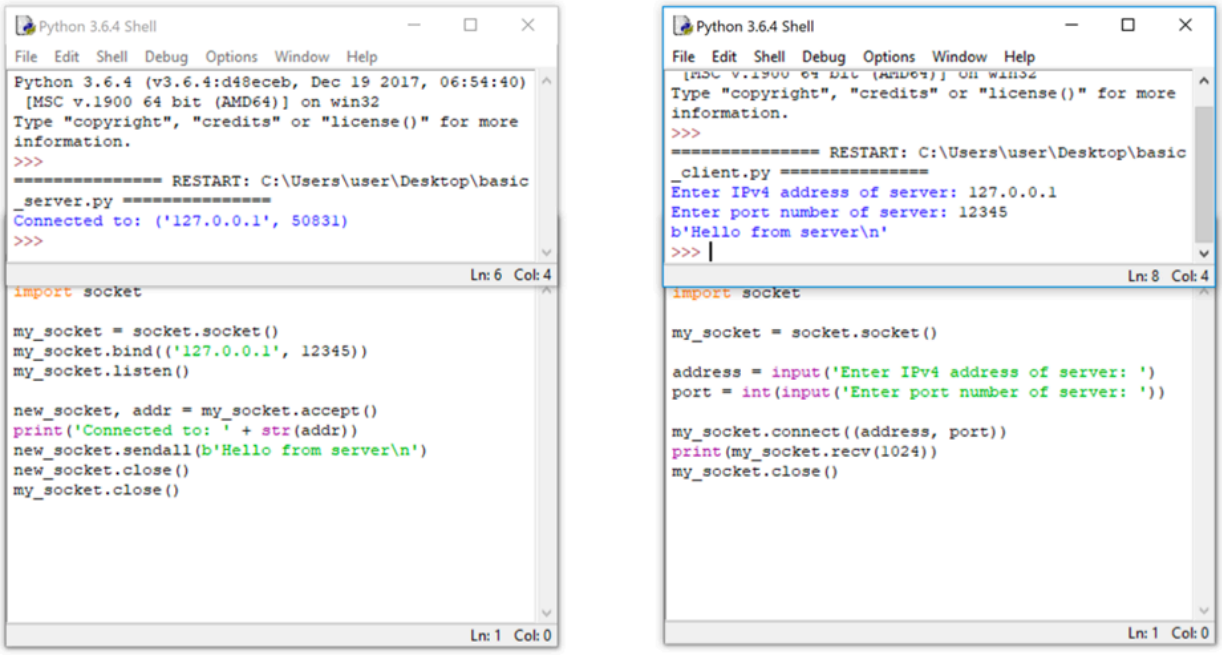
```
1   import socket
2
3   my_socket = socket.socket()
4
5   address = input('Enter IPv4 address of server: ')
6   port = int(input('Enter port number of server: '))
7
8   my_socket.connect((address, port))
9   print(my_socket.recv(1024))
1   my_socket.close()
0
```

Note that the argument for **socket.recv()** is required and should be set to a relatively small power of 2. In this case, we use a value of 2^{10} or 1024.
For more information, see:
https://docs.python.org/3/library/socket.html#socket.socket.recv

Run this program using the second copy of Python and make sure the server you started previously is still running. For instance, if you use IDLE, check that there are two shell windows running two different programs simultaneously. Otherwise, it is likely that you accidentally stopped the server when starting the client. If this happens, close the client, restart the server and make sure you reopen the client using the second shell window. If things are set up correctly, each program should affect a different shell window when it is run (e.g., by pressing F5).



At this point, the client should be prompting you for the address and port number of the server. Use the special IPv4 address 127.0.0.1 that refers to the local machine and enter 12345 as the port number. The client should successfully connect to the server and print out the bytes that were received. At the same time, the server program should become unstuck and end normally.



Congratulations, you created a socket and tried to send some data through it!

4.5 How to Design a Protocol

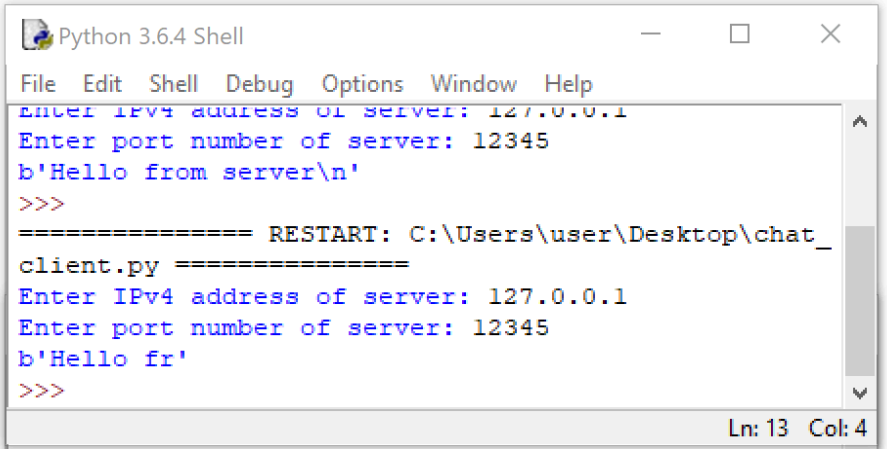
The **basic_server.py** and **basic_client.py** programs from the previous section have a hidden flaw. Observe that when using the basic server program to send longer sequences of bytes, only part of the data may be successfully transmitted even if we increase the maximum number of bytes that **socket.recv()** can receive.

Why is this so? Suppose that the sequence of bytes being sent is long enough that it needs to be sent as multiple packets. We can simulate this by breaking the sequence into two pieces and calling **socket.sendall()** twice, once for each piece. To simulate a busy network that may delay transport of the second packet, we also import the **time** module and call **time.sleep()** before sending the second piece.

Program 3: basic_server_split.py

```
1   import socket
2   import time
3
4   my_socket = socket.socket()
5   my_socket.bind(('127.0.0.1', 12345))
6   my_socket.listen()
7
8   new_socket, addr = my_socket.accept()
9   new_socket.sendall(b'Hello fr')
10  time.sleep(0.1)
11  new_socket.sendall(b'om server\n')
12  new_socket.close()
13  my_socket.close()
```

Run this version of the server, then run the client such that both programs run simultaneously on the same machine. Once again, use 127.0.0.1 for the IPv4 address and 12345 for the port number when prompted. This time, the client should receive only the first piece of data. If the client has closed the socket, the server may also produce an error when trying to send the second piece of data.



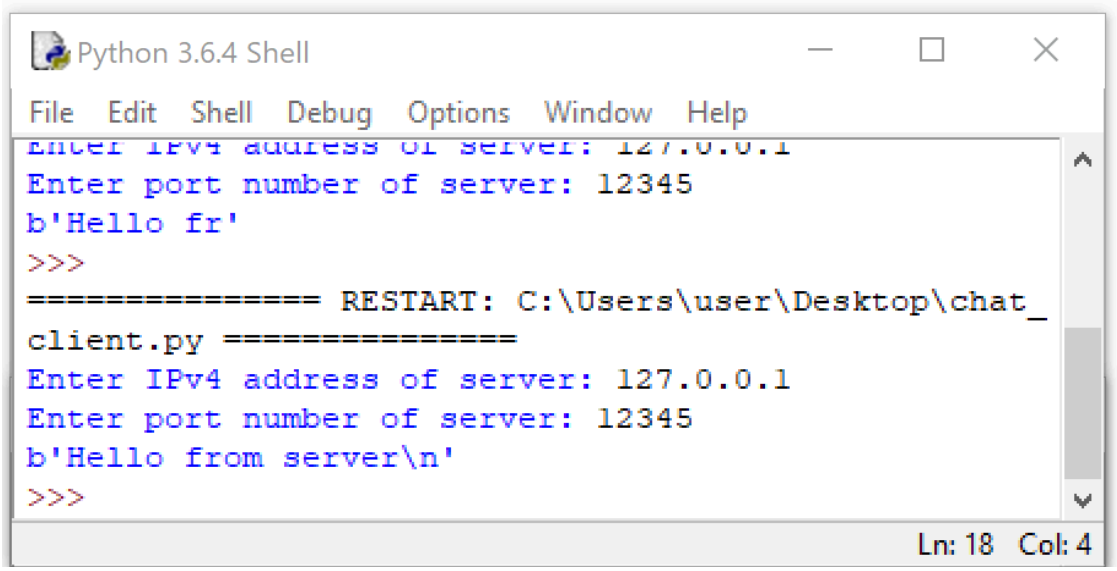
This example illustrates that, in general, we should never assume that **socket.recv()** will receive all the bytes that were sent over in one go. The only way to be certain that any received data is complete is to agree beforehand on a **protocol** or set of rules for how communication should take place. For instance, we can agree beforehand that any data we transmit will always end with a newline character **\n** and that the data itself will never contain the **\n** character. This very simple protocol allows us to detect the end of a transmission easily by just searching for the **\n** character.

The following projects update the client so that it uses the **\n** character to detect when the message ends. This new client calls **socket.recv()** continuously and appends the received bytes to a variable named data until the \n character is encountered.

Program 4: basic_client_protocol.py

```
1   import socket
2
3   my_socket = socket.socket()
4
5   address = input('Enter IPv4 address of server: ')
6   port = int(input('Enter port number of server: '))
7
8   my_socket.connect((address, port))
9   data = b''
10  while b'\n' not in data:
11      data += my_socket.recv(1024)
12  print(data)
13  my_socket.close()
```

With this new client, all the data sent by the server up to and including the \n character is successfully received and printed.



4.6 How to Create Iterative Servers

Currently, the server program exits immediately after it finishes working with a client. In reality, we often want the server program to run continuously so that it is always listening and available for multiple clients to send connection requests. We can do this by putting the code that deals with a client in an infinite loop.

Program 5: basic_server_iterative.py

```
1   import socket
2
3   my_socket = socket.socket()
4   my_socket.bind(('127.0.0.1', 12345))
5   my_socket.listen()
6
7   while True:
8       new_socket, addr = my_socket.accept()
9       new_socket.sendall(b'Hello from server\n')
1       new_socket.close()
0
```

To interrupt a program that is running in an infinite loop, press **Ctrl-C**. In IDLE, we can also restart the shell using **Ctrl-F6**.

Internally, the server's passive socket keeps a queue of connection requests that have been received. A request is removed from this queue each time **socket.accept()** is called to create a connection. If the queue is empty, **socket.accept()** will block the program until a connection request is received, as expected.

Since **socket.accept()** is called each time the infinite loop repeats, our program is able to handle multiple clients by processing them one at a time. This means that our program works as an iterative server. Iterative servers are easy to write but limited as they can only handle one client at a time.

Alternatively, we could have written our server such that it starts a thread that runs simultaneously with the main program each time a client tries to connect. This makes the program more complicated to write but will let it handle multiple clients at the same time, hence making it a **concurrent server**. In this instance, however, we will only work with iterative servers to simplify our discussion.

4.7 Writing a Chat Program

We now have all the tools needed to write a simple chat client and server such that two users can take turns sending single lines of text to each other. One user would be running the server and the other user would be running the client.

Since each message is restricted to a single line, we can be certain that the newline character **\n** will never be part of a message. This means that we can adopt a similar protocol of using **\n** to detect the end of a message.

Let us use a different port number of 6789 and create the following chat server program that repeatedly prompts the user for some text, sends that text to the client (after encoding it into bytes), then receives and prints out the client's response.

Program 6: chat_server.py

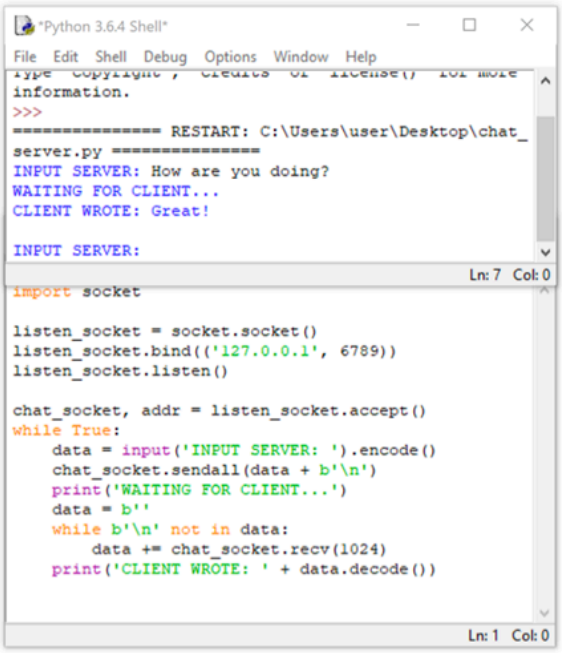
```
1   import socket
2
3   listen_socket = socket.socket()
4   listen_socket.bind(('127.0.0.1', 6789))
5   listen_socket.listen()
6
7   chat_socket, addr = listen_socket.accept()
8   while True:
9       data = input('INPUT SERVER: ').encode()
1       chat_socket.sendall(data + b'\n')
0       print('WAITING FOR CLIENT...')
1       data = b''
1       while b'\n' not in data:
1           data += chat_socket.recv(1024)
2       print('CLIENT WROTE: ' + data.decode())
1
3
1
4
1
5
```

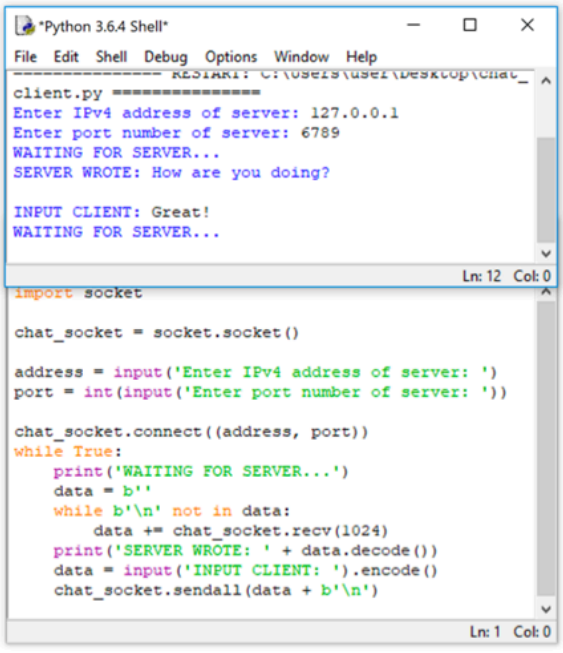
The client program is similar, except the order of sending and receiving is reversed.

Program 7: chat_client.py

```
1       import socket
2
3       chat_socket = socket.socket()
4
5       address = input('Enter IPv4 address of server: ')
6       port = int(input('Enter port number of server: '))
7
8       chat_socket.connect((address, port))
9       while True:
10          print('WAITING FOR SERVER...')
11          data = b''
12          while b'\n' not in data:
13              data += chat_socket.recv(1024)
14          print('SERVER WROTE: ' + data.decode())
15          data = input('INPUT CLIENT: ').encode()
16          chat_socket.sendall(data + b'\n')
```

Run the server and client using two different copies of Python. Once again, since the server is running on the same machine as the client, we can use 127.0.0.1 as the server's IPv4 address and 6789 as the port number.





4.8 Writing a Turn-Based Game

So far, we have been responsible for writing both the server and client programs. Sometimes, however, both server and protocol designs may be based on an existing standard or developed by someone else. To write a client that can communicate with an existing server, we need to study its code and follow the expected protocol.

Conversely, sometimes the client may be developed by someone else and we need to write a server to communicate with it. In either case, it is important to start by understanding the protocol being used.

To demonstrate how to do this, let us examine the server program for a simple turn-based 2-player game of Tic-Tac-Toe. First, we create a simple library that defines some constants and a **TicTacToe** class to handle the game logic:

Program 8: tictactoe.py

```
1   N = 3                       # Size of grid
2   WIDTH = len(str(N ** 2))    # Width for each cell
3   PLAYERS = ('O', 'X')        # Player symbols
4
5   class TicTacToe:
6
7       def __init__(self):
8           self.board = []
9           for i in range(N):
10              self.board.append([None] * N)
11
12      def render_row(self, row_index):
13          start = row_index * N + 1
14          row = self.board[row_index].copy()
15          for column_index in range(N):
16              if row[column_index] is None:
17                  cell = str(start + column_index)
18              else:
19                  cell = PLAYERS[row[column_index]]
20              if len(cell) < WIDTH:
21                  cell += ' ' * (WIDTH - len(cell))
22              row[column_index] = ' ' + cell + ' '
23          return '|'.join(row) + '\n'
24
25      def render_board(self):
26          rows = []
27          for row_index in range(N):
28              rows.append(self.render_row(row_index))
29          divider = '-' * ((WIDTH + 3) * N - 1) + '\n'
```

```
30          return divider.join(rows)
31
32      def make_move(self, player_index, cell_index):
33          cell_index -= 1
34          self.board[cell_index // N][
35              cell_index % N] = player_index
36
37      def is_valid_move(self, cell_index):
38          if cell_index < 1 or cell_index > N ** 2:
39              return False
40          cell_index -= 1
41          return self.board[cell_index // N][
42              cell_index % N] is None
43
44      def is_full(self):
45          for row_index in range(N):
46              for column_index in range(N):
47                  if self.board[row_index][
48                          column_index] is None:
49                      return False
50          return True
51
52      def get_winner(self):
53          # Check diagonals
54          if self.board[0][0] is not None:
55              found = True
56              for i in range(N):
57                  if self.board[0][0] != self.board[i][i]:
58                      found = False
59                      break
60              if found:
61                  return self.board[0][0]
62          if self.board[0][N - 1] is not None:
63              found = True
64              for i in range(N):
65                  if self.board[0][N - 1] != self.board[i][
66                          N - i - 1]:
67                      found = False
68                      break
69              if found:
70                  return self.board[0][N - 1]
71
72          # Check rows and columns
73          for i in range(N):
74              if self.board[i][0] is not None:
75                  found = True
```

```
76                  for j in range(N):
77                      if self.board[i][0] != self.board[i][j]:
78                          found = False
79                          break
80                  if found:
81                      return self.board[i][0]
82              if self.board[0][i] is not None:
83                  found = True
84                  for j in range(N):
85                      if self.board[0][i] != self.board[j][i]:
86                          found = False
87                          break
88                  if found:
89                      return self.board[0][i]
90
91          # No matching lines were found, so no winner
92          return None
```

The following is a summary of the methods in **TicTacToe** class:

Methods	Description
`render_row(row_index)`	Returns a string representation of the specified row, such as: `1 | 2 | 3`

`render_board()`	Returns a string representation of the entire board, such as:
	
	`1 | 2 | 3`
	`-----------`
	`4 | 5 | 6`
	`-----------`
	`7 | 8 | 9`

`make_move(player_index, cell_index)`	Modifies the board such that the specified cell is marked with the symbol for the specified player
`is_valid_move(cell_index)`	Returns whether the specified cell is currently blank
`is_full()`	Returns whether the entire board has been filled up
`get_winner()`	Returns winning player for the current board or None if there is no winner

Using this library, we create a server program that creates a TicTacToe object on line 9 to store information about the Tic-Tac-Toe board:

Program 9: game_server.py

```
1   import socket
2   import tictactoe
3
4   listen_socket = socket.socket()
5   listen_socket.bind(('127.0.0.1', 3456))
6   listen_socket.listen()
7
8   game_socket, addr = listen_socket.accept()
9   game = tictactoe.TicTacToe()
10  while True:
11      # Display current Tic-Tac-Toe board
12      print(game.render_board())
13
14      # Check if client player won
15      if game.get_winner() is not None:
16          print('Opponent wins!')
17          print()
18          break
19
20      # Check if board is full
21      if game.is_full():
22          print('Stalemate')
23          print()
24          break
25
26      # Prompt for move from server player
27      move = -1
28      while move != 0 and not game.is_valid_move(move):
29          move = int(input('Server moves ' +
30                           '(0 to quit): '))
31      print()
32      if move == 0:
33          game_socket.sendall(b'END\n')
34          print('You quit, opponent wins!')
35          print()
36          break
37      game.make_move(0, move)
38      game_socket.sendall(b'MOVE' +
39                          str(move).encode() + b'\n')
40
41      # Display current Tic-Tac-Toe board
```

```
42      print(game.render_board())
43
44      # Check if server player won
45      if game.get_winner() is not None:
46          print('You win!')
47          print()
48          break
49
50      # Check if board is full
51      if game.is_full():
52          print('Stalemate')
53          print()
54          break
55
56      # Receive move from client player
57      received = b''
58      while b'\n' not in received:
59          received += game_socket.recv(1024)
60      if received.startswith(b'MOVE'):
61          move = int(received[4:])
62          print('Client moves: ' + str(move))
63          print()
64          game.make_move(1, move)
65      elif received.startswith(b'END'):
66          print('Opponent quits, you win!')
67          print()
68          break
69
70  game_socket.close()
71  listen_socket.close()
```

Analysing this server code, we see that communications with the client is divided into several steps that repeat in an infinite loop:

1. Display current Tic-Tac-Toe board
2. Check if opponent has won, and if so, end game with opponent winning
3. Check if the board is full, and if so, end game with a stalemate
4. Prompt for input from player; if player makes a valid move, update game board accordingly, then send **b'MOVE'** followed by the chosen cell number and **b'\n'** to the opponent; if player chooses to quit, send **b'END\n'** to the opponent and end game with the opponent winning
5. Display current Tic-Tac-Toe board again
6. Check if player has won, and if so, end game with player winning
7. Check if the board is full, and if so, end game with a stalemate
8. Receive opponent's action via the socket; if the action is **b'MOVE'** followed by a cell number and **b'\n'**, update game board accordingly; if the action is **b'END\n'**, end game with the player winning

As written, the server player always starts first. This means that our client code should start by receiving and processing the server's result. We also know that Tic-Tac-Toe is a symmetrical game (other than the choice of starting player), so we deduce that the client code should be similar to the server code except that "client" and "server" are exchanged and the last step is moved to the front:

1. Receive opponent's action via the socket; if the action is **b'MOVE'** followed by a cell number and **b'\n'**, update game board accordingly; if the action is **b'END\n'**, end game with the player winning
2. Display current Tic-Tac-Toe board
3. Check if opponent has won, and if so, end game with opponent winning
4. Check if the board is full, and if so, end game with a stalemate
5. Prompt for input from player; if player makes a valid move, update game board accordingly, then send **b'MOVE'** followed by the chosen cell number and **b'\n'** to the opponent; if player chooses to quit, send **b'END\n'** to the opponent and end game with the opponent winning
6. Display current Tic-Tac-Toe board again
7. Check if player has won, and if so, end game with player winning
8. Check if the board is full, and if so, end game with a stalemate

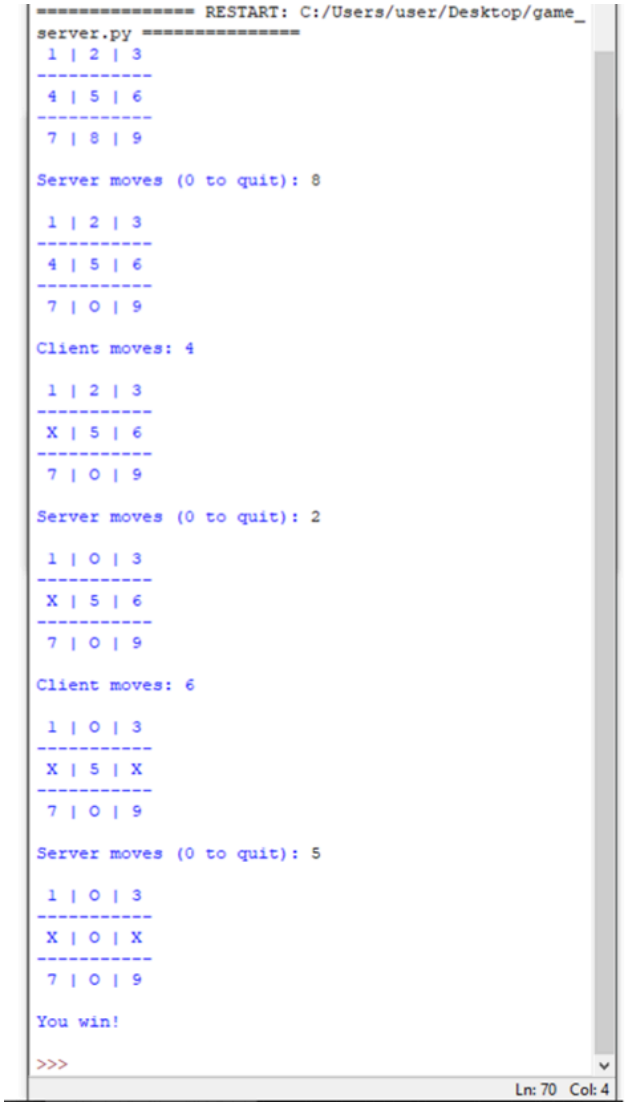
A client program that does this is as follows:

Program 10: game_client.py

```
1   import socket
2   import tictactoe
3
4   game_socket = socket.socket()
5   game_socket.connect(('127.0.0.1', 3456))
6
7   game = tictactoe.TicTacToe()
8   while True:
9       # Receive move from server player
10      received = b''
11      while b'\n' not in received:
12          received += game_socket.recv(1024)
13      if received.startswith(b'MOVE'):
14          move = int(received[4:])
15          print('Server moves: ' + str(move))
16          print()
17          game.make_move(0, move)
18      elif received.startswith(b'END'):
19          print('Opponent quits, you win!')
20          print()
21          break
22
23      # Display current Tic-Tac-Toe board
24      print(game.render_board())
25
26      # Check if server player won
27      if game.get_winner() is not None:
28          print('Opponent wins!')
29          print()
30          break
31
32      # Check if board is full
33      if game.is_full():
34          print('Stalemate')
35          print()
36          break
37
38      # Prompt for move from client player
39      move = -1
40      while move != 0 and not game.is_valid_move(move):
41          move = int(input('Client moves ' +
42                           '(0 to quit): '))
```

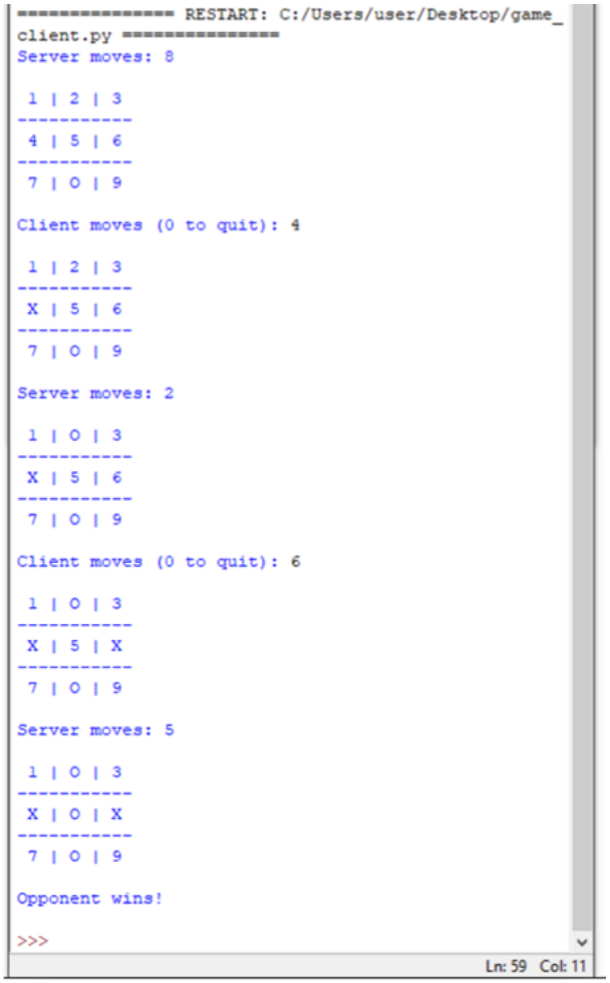
```
43      print()
44      if move == 0:
45          game_socket.sendall(b'END\n')
46          print('You quit, opponent wins!')
47          print()
48          break
49      game.make_move(1, move)
50      game_socket.sendall(b'MOVE' +
51                          str(move).encode() + b'\n')
52
53      # Display current Tic-Tac-Toe board
54      print(game.render_board())
55
56      # Check if client player won
57      if game.get_winner() is not None:
58          print('You win!')
59          print()
60          break
61
62      # Check if board is full
63      if game.is_full():
64          print('Stalemate')
65          print()
66          break
67
68  game_socket.close()
```

Run the server and client using two different copies of Python on the same machine to verify that the game works as expected. A sample run is also provided below:



```
========== RESTART: C:/Users/user/Desktop/game_
server.py ==========
 1 | 2 | 3
-----------
 4 | 5 | 6
-----------
 7 | 8 | 9

Server moves (0 to quit): 8

 1 | 2 | 3
-----------
 4 | 5 | 6
-----------
 7 | O | 9

Client moves: 4

 1 | 2 | 3
-----------
 X | 5 | 6
-----------
 7 | O | 9

Server moves (0 to quit): 2

 1 | O | 3
-----------
 X | 5 | 6
-----------
 7 | O | 9

Client moves: 6

 1 | O | 3
-----------
 X | 5 | X
-----------
 7 | O | 9

Server moves (0 to quit): 5

 1 | O | 3
-----------
 X | O | X
-----------
 7 | O | 9

You win!

>>>
```

Ln: 70 Col: 4

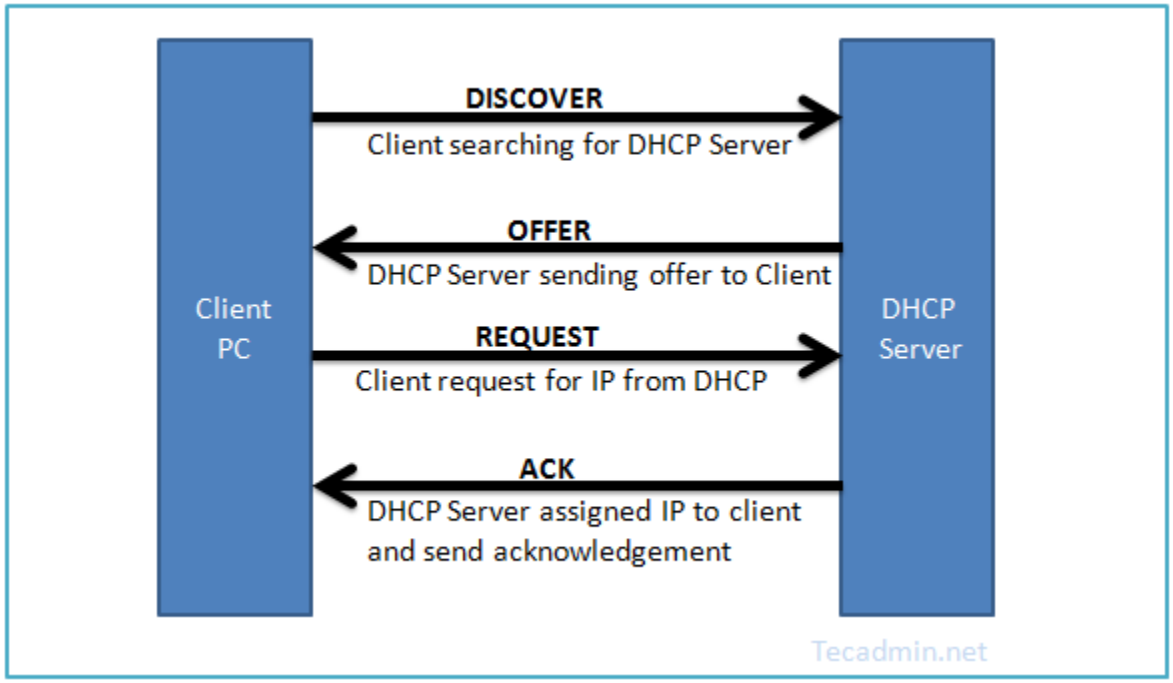


```
========== RESTART: C:/Users/user/Desktop/game_
client.py ==========
Server moves: 8

 1 | 2 | 3
-----------
 4 | 5 | 6
-----------
 7 | O | 9

Client moves (0 to quit): 4

 1 | 2 | 3
-----------
 X | 5 | 6
-----------
 7 | O | 9

Server moves: 2

 1 | O | 3
-----------
 X | 5 | 6
-----------
 7 | O | 9

Client moves (0 to quit): 6

 1 | O | 3
-----------
 X | 5 | X
-----------
 7 | O | 9

Server moves: 5

 1 | O | 3
-----------
 X | O | X
-----------
 7 | O | 9

Opponent wins!

>>>
```

Ln: 59 Col: 11

Annex - Dynamic Host Configuration Protocol

A client needs an IP address to uniquely identify itself and communicate within a network. If it does not have an IP address, the client will form an IP address request to a Dynamic Host Configuration Protocol (DHCP) server. When the server decides to allocate an IP address, it must follow a specific process known as the **Discovery, Offer, Request, Acknowledgement (DORA)** process.



First, the client has to **search** for a DHCP server. The client sends out a **broadcast message, DHCPDISCOVER,** in the form of a packet over the network. The DHCP server that is listening will then receive the broadcast message. This is how the DHCP server **discovers** the client.

Second, any DHCP on the network that receives the broadcast message will send another message, DHCPOFFER to answer the request. This is where the DHCP server '**offers'** IP addresses to the client. The new message sent by the DHCP server will contain a list of **available IP addresses** that the DHCP server can offer to the client. The message also contains other configuration parameters, such as the DNS server addresses and default gateway.

Third, the client may choose to accept or reject the IP address offer from the DHCP server. After **selecting one of the IP addresses** that the DHCP server offers, it will send a DHCPREQUEST message to the DHCP server requesting that the IP address on the DHCP server be allocated to the client domain. This message also serves as a message for other DHCP servers to know that their DHCPOFFER messages were not accepted and the offered IP addresses can be returned to their IP pool.

Finally, the DHCP server will **acknowledge** the IP address request and finalize the allocation of the IP address to the client. The DHCP server then sends out a DHCPPACK message to the

client, confirming the lease of their IP address and any other configuration details. With this final step, the DORA process of an IP address request to a DHCP server is completed.

However, if the DHCP server cannot finalise the address as it might already be assigned to another client, it will send a DHCPNACK message back to the client. The client will then have to start over again by broadcasting another DHCPDISCOVER message to the network.

Referenced Readings

1. Kurose, J., & Ross, K. (2016). Computer Networking: A Top-Down Approach (7th ed.). Pearson.

2. MOE CPDD Starter Kit: Socket Programming

3. GeeksforGeeks. Socket Programming in Python.

https://www.geeksforgeeks.org/socket-programming-python/

4. Real Python. Socket Programming in Python (Guide). Realpython.Com. https://realpython.com/python-sockets/

5. Chong, Stephen (2011). Network Programming with Sockets, CS61, Lecture 23, Harvard University, School of Engineering and Applied Sciences.