49202 Communication Protocols

The Application layer

Daniel R. Franklin

Faculty of Engineering & IT

February 28, 2023

The Application layer

- We begin our in-depth exploration of the TCP/IP protocol stack and the architecture of the Internet with the most familiar layer
- The application layer is the one that we humans directly interface with - for example
  - Browsing the web
  - Playing network games
  - Making voice/video calls (Zoom, Skype, Teams etc.)
  - Sharing files - either locally or across the Internet (e.g. NFS, SMB/CIFS, BitTorrent)

The Application layer

- It *also* includes other ‘glue’ or ‘infrastructure’ services to add useful functionality for humans - for example;
  - Domain Name System (DNS) - translates human-readable domain names like uts.edu.au to (and from!) IP addresses like 54.79.20.73
  - Directory and authentication services - e.g. LDAP, Kerberos, ActiveDirectory - for centrally managing large networks with many users, with different roles and accss rights
  - VPNs - building secure virtual networks on top of untrusted 3rd-party networks - often implemented at the application layer
  - Dynamic Host Configuration Protocol (DHCP) - automates allocation of IP addresses in a network

Classes of Application

- Applications can be classified in several different ways.
  - Based on their service requirements - for example:
    - Delay-tolerant, loss-sensitive (web traffic, file distribution, one-way video streaming)
    - Loss-tolerant, delay-sensitive (real-time two-way voice/video calls, games)
  - Based on their architecture:
    - Client-server (the traditional architecture) - e.g. a web browser and web server. This is a star topology.
    - Peer-to-peer - distributed protocols in which all hosts can play a similar role - e.g. BitTorrent, TOR. This is a mesh topology.
  - Many combinations of these classifications exist, and many applications utilise more than one mode of operation for different aspects of their functionality

Netflix and Youtube

Phone call, Zoom

Connection-oriented vs. Connectionless - Streams vs. Messages

- Depending on the service requirements of an application, two main approaches to the concept of communication have evolved in response:
  - Connection-oriented, in which there is a *persistent logical association* between two ends of the communications link, and information typically flows in both directions between the same pair of endpoints; and
  - Connectionless, in which information is sent in small, independent messages from one device to another.

Connection-oriented applications

- Connection-oriented application sessions are analogous to making a telephone call
- There is a connection setup process, in which the two ends negotiate the establishment of a connection (dialling the number, each side identifying themselves)
- If this is successful, the two endpoints enter a data transfer state - this could occur in the form of requests sent by one end to the other, which sends back responses (the actual conversation)
- Finally, one side or the other initiates a connection termination procedure, ending the session (saying goodbye and hanging up)

Connection-oriented applications

- The data transfer phase requires that information being transferred arrives intact, uncorrupted and in the correct order with no gaps:
  - This is ideal for applications which are loss-sensitive
  - However, loss/damage recovery/repair introduces some delay overhead (recovery requires resending the lost/damaged parts)
  - So this mode of communication is good for cases where delays are acceptable (delay-tolerant) - they do not degrade the integrity of the exchange.
- Logically, it is as if we have connected a wire directly from the client to the server for the duration of the connection. This may persist for seconds or longer - there is no upper limit on the duration of the connection.
- Ironically, voice calls (the original purpose of telephone lines!) are delay-sensitive and loss-tolerant - and not well-suited to connection-oriented approach!

Connectionless applications

- Connectionless applications are analogous to posting letters through the mail
- Each message is sent from a sender to a receiver independently
- They may arrive at the receiver in the wrong order - some may be missing!
- There is no persistent connection state, so there is no need for a setup or termination procedure.
  - This means we can get to the point of sending data *quickly*
  - Good where latency is the priority, i.e. delay-sensitive applications
  - *Can* be used for loss-sensitive applications - but in this case the application manages recovery/repair
  - Good for point-to-multipoint protocols, e.g. real-time video/audio broadcasting in a network

Youtube -> not Multi-cast ,

uni-cast
multi-cast
broad-cast

- Whatever **is** received gets passed immediately to the server by the operating system.
- A response *could* be sent, but it is regarded as an independent process

Services required by applications from transport / network layers

Linux , Window ..

- At this point, we should introduce a few operating system concepts. we will elaborate on these in coming weeks as we explore the transport layer. For now, we will focus on how the application actually works.
- Firstly, an important concept - the *system call*
- An *operating system* - such as Linux or Windows - runs in a *privileged mode* on the CPU of a computer system
- Applications - such as your web browser - run in *user mode*.
- When an application wants a resource, it must request this from the operating system, which can grant or deny the request depending on the privileges of the application (who is running it and what resource has been requested?
- The system call is the mechanism used by the operating system to receive such requests

Services required by applications from transport / network layers

- A system call is set up by writing the number corresponding to the type of request, and other parameters to that request, into specific CPU registers
- Then, either a special CPU instruction (e.g. `SYSCALL` on x86-64) triggers the system call
- When this occurs, the operating system's system call handler checks which system call was requested, decides whether or not it can perform the request, and if so, runs the privileged code that implements the requested service.
- When the request is completed, the results are copied into specific registers or memory locations, and the operating system hands control back to the calling application, which resumes running its own code.

Services required by applications from transport / network layers

- To make these services easier to use, your operating system provides a number of *libraries* of *wrapper functions* (subroutines with a documented programming interface) which allow these system calls to be used like normal C/C++ functions. Wrappers also exist for other languages such as Python, Rust, Java etc.
- Example: opening, writing to and closing a file in Python:

```
f = open ("output.txt", "w")
f.write ("Writing some text to a file")
f.close ()
```

- The Python interpreter will generate three system calls (open, write and close).

Operating system services for networking

- So, what is a sensible *interface* for the operating system to provide for networking services?
- In fact, communicating with a remote server via the network is much like performing file input and output (I/O)
- Therefore, the operating system uses objects called *sockets* which we interact with much as we would interact with a file.
- A socket includes several key elements. If we wish to implement a *client* that will connect to a remote *server*, we need:
  - The network-layer *address* of the remote end of the connection (either an IPv4 or IPv6 address - there are different sockets types defined for each)
  - The transport-layer *port number* of the remote end of the connection
  - The transport-layer *protocol identifier* - do we want a connection-oriented (stream) or connectionless (datagram) transport layer?

Operating system services for networking

- So the first thing to do is to construct a socket object which specifies all of these parameters
- Once it is constructed, we may then attempt to *connect* this socket to the remote server
- If this succeeds, we may then
  - Write data to the socket, which will cause the operating system to send data to the remote server
  - Read data from the socket, which will wait for the remote end to send something to us = Termination
  - Close the socket, which will free the resources and close the connection (if it is a connection-oriented stream)

Operating system services for networking

- What about on the server side? It is very similar, except now we don't specify the remote address, since we don't know who is going to connect to us in advance.
- Instead, the socket is created and *bound* to
  - An IP address of the server (which can be left blank to allow incoming connections on any server IP address); and
  - A specified TCP or UDP port on which we will *listen* for incoming connections.
- Once the socket is in the listening state, we wait for incoming connections and *block* (go to sleep!) until we receive an incoming connection attempt
- Now we can read/write/close as for the client.

Typical server-side behaviour

- There are many different ways the server can behave, depending on its purpose
- A very common approach on modern multitasking operating systems is to have one main process or thread listening for incoming connections
- When a connection is accepted, the server *immediately* creates a new *worker process or thread* and passes the socket's file descriptor to it for handling
- The main process/thread then goes back to waiting for incoming connections

Typical server-side behaviour

- The newly-created worker process/thread then can (for example) interpret requests from the remote end and perform different actions depending on the requests
- Typically it will acknowledge a request as being valid (or inform the client that it was invalid) and optionally returns some data
- When the session is finished, the process/thread will terminate the connection, free any allocated resources and exit.
- This architecture is very powerful and allows many thousands of concurrent incoming connections to a server
- Many popular servers such as the Apache web server utilise this model or some variant

Network socket buffers

- A *buffer* is just a region of memory used to store data
- It is important to realise that when data is received, it is stored by the operating system in a buffer.
- The application which is to receive this data is not obliged (의무) to read it at that moment - it can wait.
  - In this case, more data may arrive at the buffer
- Even if the application tries to read the buffer, and there is some data in it, the operating system can make it wait
  - The idea here is that if a very small amount of data has been recieved, reading this will be inefficient. So allow a bit more data to accumulate before allowing the read operation to succeed. (쌓인다)
- This behaviour can be overruled (억누르다) when you create the socket. Think about why you may want to do this!

Creating a TCP server in Python

```
import socket      # python module for TCP/IP
localAddress = "" # If blank, accept connections on any server IP address
localPort = 20001
bufferSize = 1024

msgFromServer = "Hello TCP Client"
bytesToSend = str.encode (msgFromServer)
# Create a stream (TCP) socket
TCPServerSocket = socket.socket (family=socket.AF_INET, type=socket.SOCK_STREAM)
TCPServerSocket.bind ((localAddress, localPort)) # Bind to address and IP address

# Listen for incoming datagram
while (True) :
    TCPServerSocket.listen () # Wait for incoming connection
    conn, addr = TCPServerSocket.accept() # We got one!
    message = conn.recv (bufferSize) # Read a message

    clientMsg = "Message from Client:{}".format (message)
    clientIP  = "Client IP Address:{}".format (addr)
    print (clientMsg)
    conn.sendall (bytesToSend) # Send a reply to client
```

link in learning -> python 기초 강의

Creating a TCP client in Python

```
import socket

msgFromClient = "Hello TCP Server"
bytesToSend = str.encode (msgFromClient)
serverAddressPort = ("127.0.0.1", 20001)

bufferSize = 1024

# Create a TCP socket at client side
TCPClientSocket = socket.socket (family=socket.AF_INET, type=socket.SOCK_STREAM)
TCPClientSocket.connect (serverAddressPort)

# Send to server using created UDP socket
TCPClientSocket.sendall (bytesToSend)
msgFromServer = TCPClientSocket.recv (bufferSize)
msg = "Message from Server {}".format (msgFromServer)

print (msg)
```

socket -> 라이브러리 import 하자~

Creating a UDP server in Python

```
import socket
localAddress = ""
localPort = 20001
bufferSize = 1024

msgFromServer = "Hello UDP Client"
bytesToSend = str.encode (msgFromServer) # Create a datagram (UDP) socket (below)
UDPServerSocket = socket.socket (family=socket.AF_INET, type=socket.SOCK_DGRAM)
UDPServerSocket.bind ((localAddress, localPort)) # Bind to address and ip
print("UDP server up and listening")

# Listen for incoming datagrams
while (True) :
    bytesAddressPair = UDPServerSocket.recvfrom (bufferSize)
    message = bytesAddressPair [0]
    address = bytesAddressPair [1]
    clientMsg = "Message from Client:{}".format (message)
    clientIP  = "Client IP Address:{}".format (address)

    print (clientMsg)
    print (clientIP)
    UDPServerSocket.sendto (bytesToSend, address) # Send a reply to client
```

Creating a UDP client in Python

```
import socket

msgFromClient = "Hello UDP Server"
bytesToSend = str.encode (msgFromClient)
serverAddressPort = ("127.0.0.1", 20001)

bufferSize = 1024

# Create a UDP socket at client side
UDPClientSocket = socket.socket (family=socket.AF_INET, type=socket.SOCK_DGRAM)

# Send to server using created UDP socket
UDPClientSocket.sendto(bytesToSend, serverAddressPort)
msgFromServer = UDPClientSocket.recvfrom (bufferSize)
msg = "Message from Server {}".format (msgFromServer [0])

print (msg)
```

Creating a TCP server in Python

Now waiting for incoming connections:

```
# Listen for incoming datagram
while (True) :
    TCPServerSocket.listen () # Wait for incoming connection
    conn, addr = TCPServerSocket.accept() # We got one!

    try:
        message = conn.recv (bufferSize) # Read a message
        clientMsg = "Message from Client:{}".format (message)
        clientIP  = "Client IP Address:{}".format (addr)

        print (clientMsg)

        try:
            conn.sendall (bytesToSend) # Send a reply to client
        except:
            print ("Connection is closed, can't send to client\n")

    except:
        print ("Connection is closed, can't read from client\n")
```

Creating a TCP client in Python

```
import socket

msgFromClient = "Hello TCP Server"
bytesToSend = str.encode (msgFromClient)
serverAddressPort = ("127.0.0.1", 20001)

bufferSize = 1024

# Create a TCP socket at client side
TCPClientSocket = socket.socket (family=socket.AF_INET, type=socket.SOCK_STREAM)
TCPClientSocket.connect (serverAddressPort)

# Send to server using created UDP socket
TCPClientSocket.sendall (bytesToSend)
msgFromServer = TCPClientSocket.recv (bufferSize)
msg = "Message from Server {}".format (msgFromServer)

print (msg)
```