Computer Networks
Network architecture

Saad Mneimneh
Computer Science
Hunter College of CUNY
New York

- Networks are like onions
- They stink?
- Yes, no, they have layers
Shrek and Donkey

1 Introduction (ISO OSI, IETF, and Shrek standard)

When designing complex systems, such as a network, a common engineering approach is to use the concepts of modules and modularity. In this approach, the design of the system evolves by breaking the big task into smaller tasks. Each module is responsible for a specific task and provides services to the other modules to accomplish their tasks. We can interact with a module as a black box that provides certain functionality without knowing the details of how it works. We only need to know how to interface with the module. Someone can remove the module and update it with a newer one, and we would still be able to continue our work in the same way. Moreover, modularity is important to simplify tasks (divide and conquer). For instance, in a network, reliability of message delivery and routing of messages can be treated separately by different modules. Changing one would not impact the other. If a better routing procedure is employed, it only affects the module responsible for it. Certainly, we do not desire a change in routing to affect our ability to reliably deliver messages.

Modules often interact in a hierarchy. A network is designed as a hierarchical or layered architecture in which every module or layer provides services to the upper layer. Users, sitting at the top layer of the network, communicate as if there is a *virtual link* between them, and need not be aware of the details of the network. The following figure illustrates the standardized 7 layers of a network. A description of each one follows.

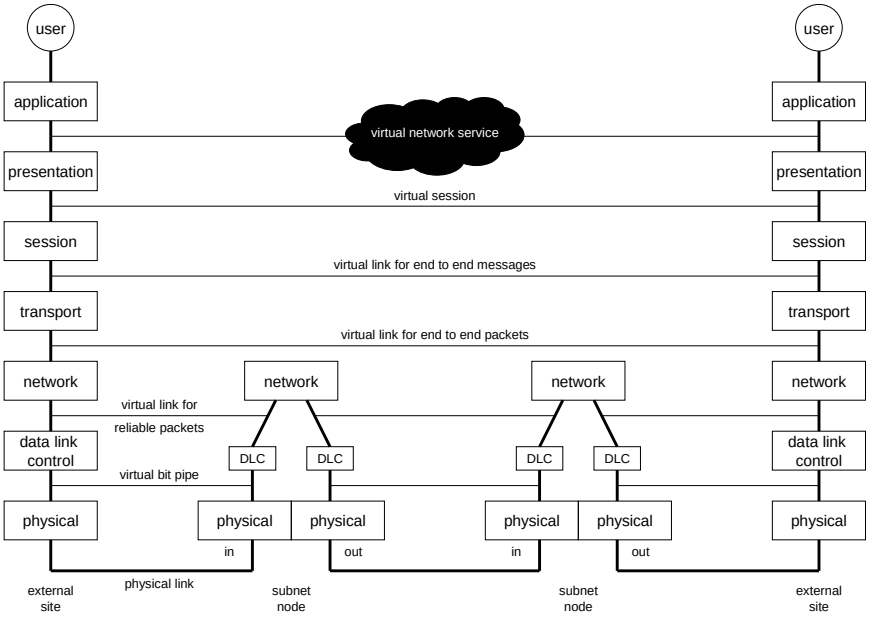

Figure 1: Network architecture: the 7 layers of each node in the network

1.1 The top layers

1.1.1 Application layer

This is the application that is used to access the network. Each application performs something specific to the user needs, e.g. browsing the web, transferring files, sending email, etc...

1.1.2 Presentation layer

The main functions of the presentation layer are data formats, data encryption/decryption, data compression/decompression, etc...

1.1.3 Session layer

Mainly deals with access rights in setting up sessions, e.g. who has access to particular network services, billing functions, etc...

There is not a strong agreement about the definition of these three top layers. Usually the focus is on the Transport layer, the Network layer, and the DLC layer.

1.2 Transport layer

While the network layer (see section below) provides end-to-end packet pipe to the transport layer, the transport layer provides end-to-end message service to the top layers.

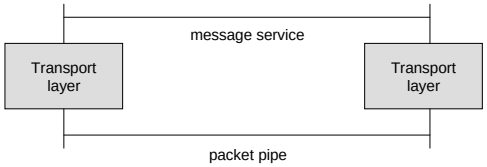

Figure 2: Transport and network layers

Functions of the transport layer include:

- Breaking messages into packets and reassembling packets into messages (packets of suitable size to network)
- Resequencing packets at destination to retrieve correct order (e.g. Datagram)
- Achieving end-to-end reliable communication in case network is not reliable, recover from errors and failures (arbitrary networks can join the Internet!)
- Flow control to prevent a fast sender from overrunning a slow receiver

Examples of transport protocols for the Internet are TCP (Transmission Control Protocol) and UDP (User Datagram Protocol). When combined with the IP protocol at the network layer, we refer to TCP as TCP/IP.

1.3 Network layer

The main function of the network layer is to route each packet to the proper outgoing DLC or to the transport layer (if the node is the destination). Typically, the network layer adds its own header (e.g. source/destination or VC number) to the packet received from the transport layer to accomplish this **routing** function.

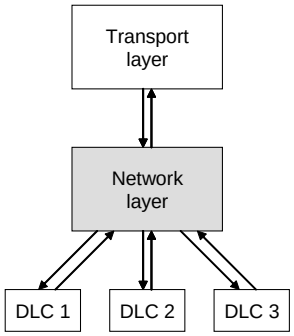

Figure 3: Routing

Headers represent a general mechanism across the layers. Each layer/protocol provides a service to the upper layer/protocol, and peer processes/protocols communicate

information through the headers. The DLC layer adds also a Trailer for error detection and correction.

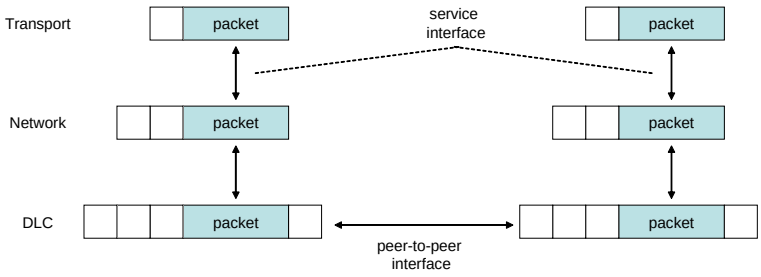

Figure 4: Headers and Trailers

1.4 DLC layer

The DLC layer is responsible for error-free transmission of packets over a **single link**. The goal is to ensure that every packet is delivered once, only once, without errors, and in order. To accomplish this task, DLC adds its own header/trailer. For instance, the header may contain sequence numbers to ensure delivery of packets in order. The packet thus modified is called a *frame*. Framing is an important issue in networks and will be discussed later.

1.5 Physical layer

The physical layer is responsible for the actual transmission of bits over a link. This layer is usually the network hardware. Higher layers, like DLC, must deal with transmission errors due to noise and signal power loss. A simple model for the physical layer is the Binary Symmetric Channel with a probability p of flipping each bit independently, i.e. $p\{0 \to 1\} = p\{1 \to 0\} = p$. However, in practice errors are bursty. There are a number of delays associated with the physical transmission:

- Propagation delay: time it takes for signal to travel from one end of link to another = distance/speed of light
- Bandwidth: number of bits that can be transmitted over a period of time, i.e. bits per second (bps)
- Latency of packet = Propagation delay + size of packet/Bandwidth + Queuing delay
- RTT = Round Trip Time for exchanging small messages $\approx$ 2(Propagation delay + Queuing)

2 The hourglass shape (an alternative view)

Despite the hierarchical nature, Internet architecture does not imply strict layering. Many times there are multiple protocols provided at a given layer in the system, giving

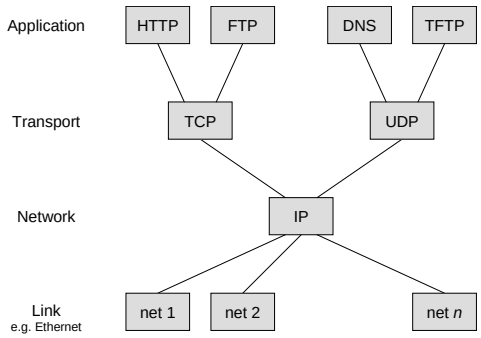

Figure 5: Hourglass shape

an hourglass shape to the architecture. An application is free to bypass the defined transport layers and to directly use IP.

IP serves as a focal point (common method for exchanging packets among networks): many transport protocols lie above IP (define services), and many networks technologies lie below IP.

3 Network software

Most network protocols are implemented in software (that's one of the main reasons for the Internet's success). All computer systems implement their protocols as part of the operating system. The operating systems exports the network functionality as a Network Application Programming Interface (API). Some API is widely accepted and supported: the *socket interface*.

The main abstraction of the socket interface is the *socket*: a point where the local application process attaches to the network. The API specifies ways to

- create sockets
- attach the socket to the network
- send/receive through the socket
- close the socket

3.1 Create a socket

The following C function creates a socket:

```
int socket (int domain, int type, int protocol)
```

The parameters are as follows:

- **domain**: specifies a protocol family
  - PF_INET denotes the Internet family
  - PF_UNIX denotes the Unix pipe family
  - PF_PACKET denotes direct access to the network interface, i.e. bypass TCP

- **type**: specifies the semantics of the communication
  - SOCK_STREAM denotes a byte stream
  - SOCK_DGRAM denotes a message oriented service
- **protocol**: identifies a specific protocol to be used, in our case we can simply ignore it because the combination PF_INET and SOCK_STREAM implies TCP
- The return value is an identifier of the newly created socket

3.2 Server

The following C functions are often used by a server:

```
int bind(int socket, sockaddr * address, int addr_len)
int listen(int socket, int backlog)
int accept(int socket, sockaddr * address, int * addr_len)
```

The **bind** operation binds the socket to the network address, a data structure that includes

- IP address
- TCP port (a port is used to identify the process)

The **listen** operation defines how many connections can be pending on a given socket.

The **accept** operation is a blocking operation that does not return until a remote participant has established a connection. It returns a new socket that corresponds to just this connection, and the address parameter will contain the remote participant's address.

3.3 Client

The following C function is often used by a client:

```
int connect(int socket, sockaddr * address, int addr_len)
```

The connect operation does not return until TCP has successfully established a connection. The address will contain the remote participants (server) address. The client does not typically care which port it uses, OS simply selects an unused one. Once a connection is established, the client can use the following functions to send and receive data:

```
int send(int socket, char * message, int msg_len, int flags)
int recv(int socket, char * buffer, int buf_len, int flags)
```

3.4 Example server

```
#include <iostream>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <unistd.h>

using std::cout;
using std::cin;

const unsigned short int port = 5432;
const int max_pending = 1;
const int max_len = 256;

int main() {

  sockaddr_in address; //address
  sockaddr_in client_address; //client address
  char message[max_len];

  int s;
  int new_s;
  int len;

  //build address
  bzero((char *)&address, sizeof(address));
  address.sin_family = AF_INET;
  address.sin_addr.s_addr = htonl(INADDR_ANY);
  address.sin_port = htons(port);

  //setup passive open
  if ((s=socket(PF_INET, SOCK_STREAM, 0)) < 0) {
    cout<<"error in socket";
    return 0;
  }

  //bind socket to address
  if (bind(s, (sockaddr *)&address, sizeof(address)) < 0) {
    cout<<"error in bind";
    return 0;
  }

  if (listen(s, max_pending) < 0) {
    cout<<"error in listen";
    return 0;
  }

  //wait for connection, then receive message
  socklen_t size;
```

```
  while (1) {
    if ((new_s = accept(s, (sockaddr *)&client_address, &size)) < 0) {
      cout<<"error in accept";
      return 0;
    }
    while (len = recv(new_s, message, sizeof(message), 0))
        cout<<message<<"\n";
    close(new_s);
  }
}
```

3.5 Example client

```
#include <iostream>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <unistd.h>

using std::cout;
using std::cin;

const unsigned short int port = 5432;

int main() {
  int s;
  sockaddr_in address;

  bzero((char *)&address, sizeof(address));
  address.sin_family = AF_INET;
  address.sin_port = htons(port);
  address.sin_addr.s_addr = htonl(INADDR_ANY); //my IP address


  //active open
  if ((s=socket(PF_INET, SOCK_STREAM, 0)) < 0) {
    cout<<"error in socket";
    return 0;
  }

  //connect
  if (connect(s, (sockaddr *)&address, sizeof(address)) < 0) {
    cout<<"error in connect";
    close(s);
    return 0;
  }
```

```
  char message[256];
  while(cin.getline(message, 256, '\n')) {
    if (strlen(message) == 0)
      break;
    send(s,message,strlen(message)+1,0);
  }
  close(s);
}
```

References

Dimitri Bertsekas and Robert Gallager, Data Networks
Larry Peterson and Bruce Davie, Computer Networks: A Systems Approach