Lecture 2

Wireshark Project 1 posted, due in a week
Lab from a different textbook
Work through the lab and answer questions at the end

Outline

Chapter 1 - Foundation
1.1 Applications
1.2 Requirements
1.3 Network Architecture
1.4 Implementing Network Software
1.5 Performance
1.6 Summary

Network Architecture

To deal with the complexity of network design, engineers have developed network architectures that are used to design and implement networks.

After discussing general network architecture concepts we will examine the two most popular architectures – the OSI and Internet architectures.

Layering and Protocols

Network architectures are defined in terms of abstract layers.
Layering decomposes the design into simpler components. It also allows for a modular design.

Layering and Protocols

Network architectures will typically have multiple abstractions at any given level. The abstract objects are known as protocols.
A protocol provides a communication service that higher level objects use to exchange messages.
Layering and Protocols

Protocol specifications define the service and peer interfaces.

- The service interface defines the operations the protocol provides to other local, higher-level objects.
- The peer interface defines the form and meaning of data exchanged with the peer protocol on another machine.

Layering and Protocols

Higher-levels encapsulate data in a new message (typically by adding a header) as data is passed down the protocol stack.

The headers are removed as the data passes up the stack on the peer.

Layering and Protocols

Multiplexing and demultiplexing can be achieved through the use of identifiers (demux keys) in the header.

For example, different applications would be assigned different keys by RRP. The key is used on the peer to direct the message to the correct corresponding peer application.

Similarly, keys can be used by HHP to demux a message between RRP and MSP.

OSI Architecture

The OSI architecture is one of the oldest. It consists of seven layers. Refer to the text for details.

TCP/IP Architecture

The Internet (TCP/IP) architecture is overwhelmingly the most popular architecture in use. It has made almost all other architectures obsolete.

The TCP/IP network model consists of four layers.
TCP/IP Architecture

TCP/IP can utilize many different Network protocols (Ethernet, token-ring, serial line, carrier pigeon). The Network layer is typically implemented using hardware (a network adapter) and software (a device driver).

The IP or Internet Protocol defines how messages are routed between networks. It provides support for connecting networks to form an internetwork.

TCP/IP Architecture

The Transport layer (above the IP layer) consists of two main protocols: TCP (Transport Control Protocol) and UDP (User Datagram Protocol). TCP is a reliable, connection-oriented channel (similar to our theoretical RRP protocol) while UDP is an unreliable, connection-less channel (similar to MSP).

The top or Application layer consists of many, many protocols (FTP, Telnet, SSH, SMTP, HTTP, etc). Programmers often define their own protocols for comm. between applications.

TCP/IP Architecture

The TCP/IP model has three notable features:

Layering is not strict. Applications can bypass the Transport layer and use the IP or Network layers.

The model is narrow-waisted. The IP layer can utilize multiple Network protocols. Many Transport protocols can be implemented above IP.

The protocol suite can be expanded by adding new protocols. Official inclusion requires a specification AND an implementation.

Implementing Network Software

The success of the Internet can be attributed to the fact that much of its functionality is provided by software running on general-purpose computers.

Knowing how to implement network software is essential to understanding computer networks. The way an application uses the network is similar to the way in which a high-level protocol uses a lower-level protocol.

The socket API

The socket interface to the TCP/IP protocol suite is supported on all popular OSes. Porting network applications that use the sockets API is relatively simple.

The socket is the main abstraction in the socket API. Think of a socket as the point at which an application connects to the network.

The socket API

To create a socket in C/C++ call the socket function:

\[ \text{int socket(int domain, int type, int protocol)} \]

Note: The socket API supports communication over several network protocol suites (TCP/IP, IPX, Appletalk, etc) as well as communication between processes on the same machine (local sockets).
The socket API

To specify that we want to use the IP protocol, domain should be `AF_INET`. (To bypass IP and use raw packets use `AF_PACKET`.)

The type argument should be either `SOCK_STREAM` to specify TCP or `SOCK_DGRAM` to specify UDP.

The protocol argument should be 0 (zero) when the domain is `AF_INET`.

An integer identifier (the socket) is returned. It is used in all subsequent socket functions.

The socket API - Server Side

If you are writing a server that uses the TCP transport protocol, you would then call:

```c
int bind(int sck, struct sockaddr *addr, int addrlen)
int listen(int sck, int backlog)
int accept(int sck, struct sockaddr *addr, int *addrlen)
```

Each of these functions takes a `sck` argument, this is the identifier that is returned by a previous call to `socket()`.

The backlog argument to `listen()` defines the maximum length to which the queue of pending connections for the socket may grow.

The server process will block at `accept()` until a client connects.

The `addr` argument is a pass-back argument, it contains the IP address and TCP port number of the client.

The `addrlen` argument is a value-return argument. It should be initialized to the size (in bytes) of the `addr` structure.

The socket API - Server Side

A server application waits for clients to connect to a particular socket address on the local machine. It does a passive open of the socket.

The `addr` argument used in the `bind()` call is a pointer to a data structure that contains the IP address of the server and a TCP port number.

The TCP port number is used as a demux key to associate received data with this server.

The `addrlen` argument is the size of the `addr` data structure in bytes.

The socket API - Client Side

After creating a socket via a call to `socket()`, the client performs an active open via:

```c
int connect(int sck, struct sockaddr *addr, int addrlen)
```

The `addr` argument contains the IP address and port number of the server the client wishes to connect to.

On error, -1 is returned. An error may occur due to network problems or if no server is listening at the indicated `addr`.

The accept() routine returns an index to a new, active socket. This new socket is used for communicating with the connected client.

In the event of an error, each of these routines (`socket()`, `bind()`, `listen()`, `accept()`) returns the value -1. You should ALWAYS check for an error and take appropriate action.

Refer to the man pages for each of these routines for additional details.
The socket API - Communication

After a connection had been established data is transferred between the server and client using:

- `int send(int sck, void *buf, size_t len, int flags)`
- `int recv(int sck, void *buf, size_t len, int flags)`

`recv()` will block until a message is available or the connection is terminated. The `write()` and `read()` routines can used instead of `send()` and `recv()`.

Note that sockets are bidirectional.

In-class Exercises

- After discussion of the simplex server and client, compile the client and connect to the server being run at csserver.
- Modify the server to display the IP address and port being used at the client peer (`getpeername()`). Run this server on localhost with a localhost client using hostname "localhost".

In-class Exercises

Use a port number of 0 on the server, so that a port is dynamically assigned by bind(). Use `getsockname()` to determine and display the port number. Modify the client to accept the server port number as the second argument. Run the server on csserver; run the client on localhost.

When you have complete this exercise, zip the client and server files together and email the archive as an attachment to the instructor at randall@evansville.edu