Notes for ECE 46300 - Introduction To Computer Communication Networks

Zeke Ulrich

September 10, 2025

Contents

Course Description

An introduction to the design and implementation of computer communication networks. The focus is on the concepts and the fundamental design principles that have contributed to the global Internet success. Topics include: digital transmission and multiplexing, protocols, MAC layer design (Ethernet/802.11), LAN interconnects and switching, congestion/flow/error control, routing, addressing, performance evaluation, internetworking (Internet) including TCP/IP, HTTP, DNS etc. This course will include one or more programming projects.

Computer Networks

The high-level question this course will answer is "how do computers reliably communicate?"

The answer is through computer networks, a group of interconnected nodes or computing devices that exchange data and resources with each other.

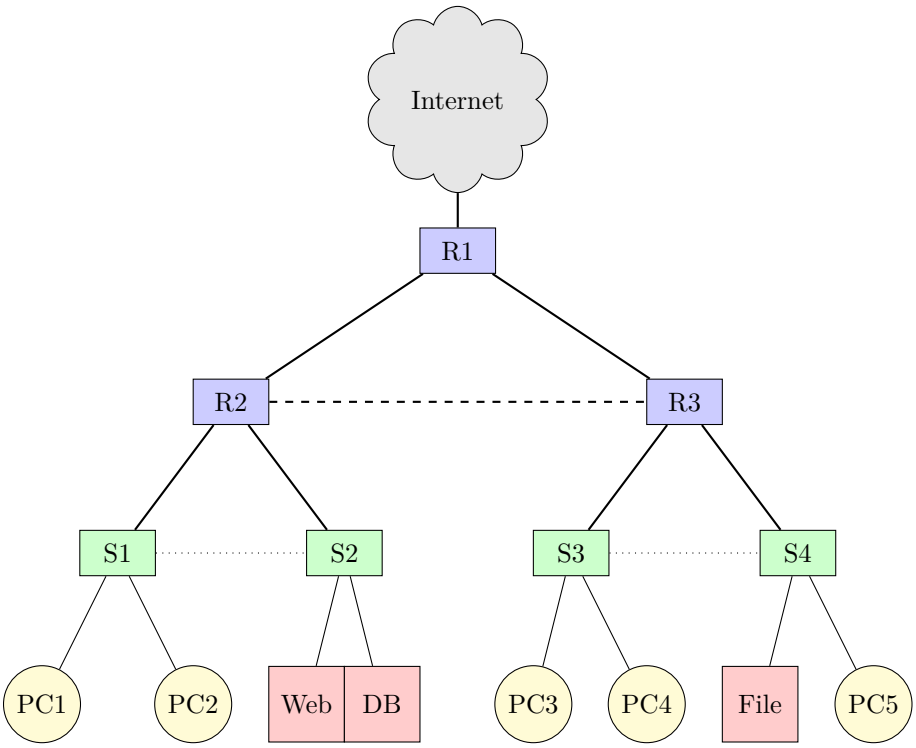

Figure 1: Computer Network

A computer network enables communication between users and their devices.

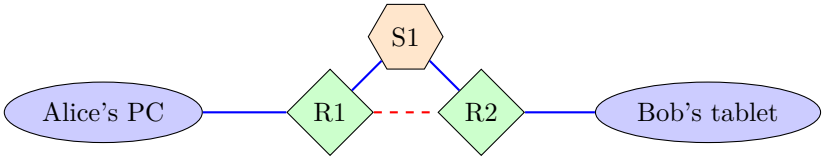

Figure 2: Simple Network

The first stop that most home devices take in connecting to other devices is the router. From there, the router can route data and find a path for Alice's PC to get to Bob's tablet. The core of any network is routers, which figure out the best way to get data from one device to another device.

This process can get complex, with cell tower connections, different ISPs, different edge devices, and more. Let's abstract the important elements of the network.

- Links: carry data from one endpoint to another
- End hosts: sitting at the edge of a network. Generate and receive data.
- Routers: forward data through the network.

Any network can be abstracted as a connection of links, end hosts, and routers. We can thus represent any computer network as a graph, in the mathematical sense, and apply all our graph algorithms to it.

Packets

In our abstraction of routers, we leave the problem of finding a path from end hosts unanswered. Since we can represent a computer network as a graph, a shortest-path algorithm like Dijkstra's is a relatively simple way to find a good path between Alice and Bob.

We wish to answer the question "how do users access shared network resources?" Assume no coordination between users and assume users initiate access.

There are two ways to answer this:

- Reserving resources (circuit switching).
- On-demand (packet switching)

Circuit Switching

The idea at the core of this method is that if Alice wants to communicate with Bob, then they reserve a path through the network to do that. The reserved path is called a *circuit*. They then send data along the reserved path.

A pro of circuit switching is that users are guaranteed to have a path through the network. No queuing, no waiting. The routers along the path also don't need to make any decisions in real time. They know the path in advance.

A con is that a resource could be reserved but not used. When users don't coordinate, circuit switching leads to very inefficient use of resources. Another con is that circuits need to be set up and torn down. Imagine the overhead of old telephone networks, where workers had to manually connect telephone wires from one caller to another.

In classical circuit switching, all the resources along the path are reserved for a single circuit and there is no path sharing amongst multiple circuits. In virtual circuit switching, each circuit reserves a subset of resources along its path. It's still reservation based, but two or more circuits can share a same resource (e.g. if two users want to send data that takes up half the bandwidth of a connection, they can both send it at once).

Packet Switching

To overcome the shortcomings of circuit switching, packet switching was invented. The way this works is data is broken into small units called *packets*. You send packets over the network whenever you have them and trust the routers to figure out the path your packets take on the fly.

Each packet needs to have metadata that describes things like the destination of the packets, the order of the packets, and so on.

A pro of packet switching is that we have much better resource utilization. Also, there's no overhead of circuit setup and teardown.

A con of packet switching is that there's no guarantee that a given user will have the resources to send their data. Now we also have overhead from the packet headers, and routers now need to process packet headers and find paths on the fly.

Although both have advantages and disadvantages, modern networks almost exclusively use packet switching and unless otherwise specified all networks in this class are assumed to be packet switched.

Packet Headers

Packet headers require, at least, the following:

- Destination address, used by network to send packet to destination
- Destination port, used by network stack at the destination for application multiplexing
- Source address, used by network to send packet back to source
- Source port, used by network stack at the source for application multiplexing

We'll learn more about the OSI model in a later section, but for now suffice to say that each layer has its own header that it stacks onto and strips from user data, starting with the application layer attaching its own header to user data, to the transport layer sticking a header on, and so on. As the package travels from host to destination the layers get added, and then stripped off in the order they were added.

Internet Architecture

We need to solve these problems:

- Name and addressing: identifiers for network nodes
- Destination discovery: finding the destination address
- Forwarding: sending received data to the next hop (neighbor)
- Routing: finding the path from source to destination
- Reliability: handling failures, packet drops, packet corruption
- Application multiplexing: delivering data from multiple host applications to the network and vice versa

Name and Addressing

Name is the human-readable name for each node (e.g., URL www.google.com). Address is where node is located (e.g., IP address 172.21.4.110).

Destination discovery

When you go to a web page, you type the name in your browser. You need to get the address still. The way names are resolved into addresses is via the Domain Name System (DNS), a set of global servers that maintain the mappings between host names and addresses. When you type a URL, the browser contacts a DNS server to get the address.

Forwarding

A router has many ports. Each port in a router acts as both input and output, i.e., you can both send and receive packets on each port simultaneously. When a packet arrives at a router, the router looks at the destination address in the header. Based on destination address, the router consults the routing table, determines the right output port, and sends the packet to that port's queue. For each output port in parallel, when the port is free, the router picks a packet from the corresponding output queue in some order (e.g. FIFO) and sends the packet over the output port.

Routing

How do a network of routers collectively find a path between each source and destination host? The answer is a routing protocol, a distributed algorithm that runs independently at each router.

Reliability

The goal of reliability is to ensure every packet sent will eventually reach the destination uncorrupted. Unfortunately, routers can drop packets or packets can get corrupted. The idea behind all reliability protocols is after sending data, await an ACK from the destination. If ACK was received, everything is good! If not, resend the data.

Application Multiplexing

Each app that communicates over a network needs some subset of basic functionalities, like a reliability protocol. We could ask each app developer to implement the protocol themselves, but this is burdensome on the developer and prone to errors. Instead, on modern computers, there is a common OS service that handles packing data into packets, creating packet headers, and handling ACKs.

Typically, each host has a program running inside the OS called "host network stack". In addition to the responsibilities above it also handles getting data from the network and sending it to the right application.

When an app wants to access the network, it opens a *socket* which is associated with a *port*. A socket is an OS mechanism that connects applications to the network stack. A port is a number that specifies a particular socket. The port number is used by the OS for app multiplexing to direct incoming packets to the right applications.

Modularity

The internet is inherently hierarchical. There are big networks at high levels, e.g. AT&T, which connect smaller networks at a lower hierarchy e.g. Purdue. We break the task of sending data into five layers, known as the *Open Systems Interconnect* (OSI) model.

1. Physical: bits sent over a physical connection. Involves signal processing, analog-to-digital conversions, etc.
2. Data Link: "best-effort" data delivery within a local network. Involves naming, addressing, destination discovery, routing, forwarding
3. Network: "best-effort" data delivery between networks. Involves naming, addressing, destination discovery, routing, forwarding
4. Transport: reliable end-to-end data delivery. Involves reliability and application multiplexing.
5. Application: network service to users or applications. Involves read/write data from applications, encrypt/decrypt, etc.

Each layer only talks with the layers directly above and below.
Where should these subtasks be implemented?

Sockets

A *socket* is an OS mechanism for inter-process communication. It allows two processes or applications on the same (loopback) or different machines to communicate with each other. The communication path goes through the network stack of machines running the processes.

The socket interface sits between the application and transport layer. It has send and recieve buffers for each app, which apps access via `send()` and `recieve()`. Each application is attached to a socket that has a unique port number and send-recieve buffers.

Connection

A *connection* is identified with a 5-tuple:

- Local IP address
- Remote IP address
- Local port
- Remote port
- Transport protocol type

These 5 things in combination make a connection. A local and remote socket make a connection.

Any address that begins with `127.` is on the local machine, i.e. on loopback.

Creating a socket is the first step in socket programming. Use the `socket()` system call, which returns a file descriptor. This is interesting, because it abstracts the difficulties of sending data over a network to the same process as writing to a file.

There are three things needed to open a socket, which is done with a call like `int fd = socket(int family, int type, int protocol)`

- `family`: protocol family used for communication. E.g. `AF_INET`.
- `type`: defines the communications semantics used. There are two popular choices, `SOCK_STREAM` which is reliable, and `SOCK_DGRAM` which is unreliable.
- `protocol`: specifies a particular transport protocol. Setting to 0 will choose the default protocol implemented in the OS. E.g. TCP, UDP.

SOCK_DGRAM is connectionless, i.e. no handshake required before sending data. This is a packet or *datagram* abstraction. SOCK_DGRAM offers unreliable communication in the sense that there is no promise that every packet is delivered. For datagram abstraction, any data sent with `send()` just has a UDP header slapped on and that's sent off as a packet. Whatever transport layer on the other side just strips off the UDP header and forwards the payload to the recieving app.

SOCK_STREAM is connection-oriented, i.e. explicit handshake happens before data is sent. This is a byte stream abstraction. SOCK_STREAM offers reliable communication, which means that all the data reaches its destination reliably and in-order. For bytestream abstraction, there is a buffer. Bytes sent are stored in the buffer. The TCP protocol decides how many bytes go into packet.

In either case, excess bytes are dropped if the receive buffer is full. In the datagram case nothing is done. In the bytestream case, the sender is notified and presumably resends the bytes.

In the datagram abstraction, if the sending app makes five send calls, the receiving app should make five receive calls. Nothing similar is promised in the bytestream abstraction.

Note that the actual data going over the network is packets, regardless of if a byte stream abstraction or datagram abstraction is used. The abstraction is just for programming convenience.

A recap of important function calls:

- `socket()`: creates a socket
- `bind()`: bind socket to an address <IP, port>
- `connect()`: initiate connection to server
- `listen()`: listen for and queue incoming client connections
- `accept()`: accept incoming client connections
- `send()`: send data over a connected socket (TCP)
- `recv()`: receive data from a connected socket (TCP)
- `sendto()`: send data to a specific address (UDP)
- `recvfrom()`: receive data and sender's address (UDP)
- `close()`: close the socket and release resources