CSCI 3171

Introduction

McAllister

Alex Safatli

Monday, January 7, 2013

Contents

Introduction

Administration

The tutorial for this class is optional – it is used to supplement material from class. The learning outcomes are explicitly written – the verbs are important and identify what we need to know.

For the protocols we will look at, we will come to understand them and analyze how they work. We will investigate what parts can be used elsewhere. The fourth-year counterpart to this will look at how these work at a low-level, but we will investigate how these networks at a higher-level.

Assignments before this year have been predominantly written, but this year will be a broken-up programming problem meant to implement a Dropbox-like networking component. These sort of milestones will not be worth any marks, but are meant to be ways for us to pace ourselves. However, they can still be submitted and feedback can be given. Tentative dates for assignments are given on the syllabus.

The purpose of the project is intended to be an exploration of a network concept. A set of ideas are presented but are not the final list – you can choose anything you would like. The Friday after reading break, the topic should be indicated and narrowed or expanded. They can be done individually or in groups of two. Furthermore, the five assignments are due throughout term and the project will need to be done in parallel.

Programming can be done in any language you are familiar with (for example, Python, C, Java, etc.). Code will be handed in through SVN (see syllabus). Anything written will be submitted through Moodle (`http://courses.cs.dal.ca`).

Networking & the Internet

The **Internet** is an interconnection of hosts (end nodes) and network devices (routers, switches, WAPs, etc.). It is connected by communication links and is a loosely hierarchical network of networks. *Locality* ensures not everything is distributed everywhere. There are also often a number of types of networks we can talk about:

1. **Internet**: Worldwide open interconnection of public networks.
2. **Intranet**: Interconnection of networks within a company. Tend to be closed.
3. **Extranet**: Interconnection of networks within a company with a few computers from trusted partners connected via secured links. An extension of an intranet. Example: Walmart – triggers are set-up so that orders can be made as soon as supplies go low; VPN clients.

What makes these networks work? **All the communication is managed through protocols**. A protocol defines the *format* and *order* of messages sent between network entities, and the *actions to take* when they are sent or received. Adherence to the protocol promotes INTEROPERABILITY and ensures communication is happening properly.

Networks rely on cooperation between network entities. Protocol compliance is not centrally certified. What incentive do network entities have to cooperate? This is interoperability. If you do not follow protocols, the rest of the network will not try to cooperate with you and will not want to have anything else to do with you. See the slide on *finite state machines* and how transitions are made. For example, FIN/ACK means if FIN is received, send ACK. An example of a protocol is UDP.

Who defines these protocols? Internet standards exist (IETF through Requests for Comments OR IEEE), as well as web standards (W3C), and general standards (ISO AND ITU). Most of these are on volunteer basis. The web as we know it only came out shortly after the beginning of networkings (UDP was drafted in the

Networking & the Internet continued on next page...

80s), defined by HTTP, and this was around the same time as the formation of W3C. Could you ask *who owns the internet*, though? Or who *controls* it? One level of control is through these protocols. Furthermore, to a degree, the US Department of Defence essentially started the Internet, funds the ICANN corporation to handle names, and the bulk of computers to manage domain names started there as well. Note that between two different protocols, such as one defined by the IETF and one by the ISO, the former is founded more in engineering.

What does the Internet do? It offers communication services. The type and complexity of the services varies:

- Recognize a bit on a wire.
- Send a message across a cable between two computers.
- Find a path to an arbitrary computers on the network.
- Send a message reliably to an arbitrary computer.
- Trade files across a network.
- Make another computer seem like an extension of my own.
- Coordinate distributed applications.

What kind of issues can arise in networks? There may be malicious users. The signal may require security and has to be understood. There may need to be management of a high or low load – collisions may occur, who gets to send, and what do you do with lost messages? Lastly, structures of the network changing could be an issue that has to be handled.

How do we deal with network complexity? Recall lessons from programming:

- modularity,
- data hiding,
- separation of concerns,
- application programming interfaces (APIs),
- best practices (already out there),
- and standardization.

Furthermore, create **reference models** or organize the complexity. That is to say, divide the network tasks into **layers** and isolate functionality to specific layers. The Open System Interconnection (OSI) Reference Model is defined the ISO and is often used for discussions, but is not the dominant model for the Internet architecture.

- Application,
- presentation,
- session,
- transport,
- network,
- link,
- and physical.

The operating system typically handles the transport layer and those below it. The physical medium is your wire, wireless system, etc. If a particular layer has to be changed, the rest of the system should still function.

The **application layer** provides communication services in direct support of user tasks. This includes SMTP (for e-mail), HTTP, BitTorrent, etc. It is not the same as the program running the protocol. This is embedded in the network program.

The **presentation layer** ensures that communication hosts can properly recognize each others' information. It deals with byte order, string encoding character set, collapsing data structures, and data encryption.

The **session layer** provides continuity of conversations or a dialogue across multiple messages. For example, remote procedure calls.

The **transport layer** handles all concerns of reliable data transmission. It deals with data integrity, guaranteed delivery, ordering of messages, flow control (speed), and congestion control (the volume of all messages together). Anything below this layer is designed to be unreliable.

The **network layer** delivers messages between any two hosts on the network, and they may not necessarily be directly connected. For example, it handles network links that fail.

The **link layer** delivers messages between two directly-connected hosts on the network. It picks out messages from a stream of bits.

The **physical layer** encodes bits onto a physical medium. For example, fibre optics, wireless transmission, etc.

The TCP/IP Reference Model, defined by the IETF, defines the architecture of the Internet. It conceptually collapses the application, presentation, and session layers from the OSI model into one application layer. The other layers remain the same.

The link and physical layers are typically hardware (network card) and are closely bundled together. The IETF typically handles these. The OS, on the other hand, handles the network and transport layers, typically. However, sometimes the network layer can be embedded in hardware. These can be, for instance, network daemons. Finally, the application model of the TCP/IP model is usually the network program itself, in addition to software libraries.

NOTE Personal computers connected directly in a LAN and getting internet down from regional ISPs – see image – are known as *network edges*. ISPs and more specialized inner links are *network cores*.

Encapsulation is, by definition, to enclose in or as if in a capsule. This is our notion of *data hiding*. When a layer sends a message through the lower layer, the lower layer prepends a header to the message to store any information relevant to that layer (for instance, where it is heading). It does not change any information in the message that it receives. In the opposite directions, layer headers are removed before passing the message on to the next higher layer. When layers changed, those headers are all that are changed.

Furthermore, all layers in each network device may differ and may implement a different set of them, such as in a switch or router. The only layers that will be able to read or remove this header are the same sort of ones that applied these headers. See slides for more information.

We tend to use different names for messages depending on what level of the stack they are associated with (those from the application layer are messages or protocol data unit; *PDU*). When it hits the transport layer, for instance, it becomes a *segment*. In the network layer, we typically call these *packets*. The link layer messages are called *frames*.

One of the things we will deal with are two concepts: **multiplexing** and **multiplexing**. When you merge information from several sources into a single channel, this is known as *multiplexing* (many protocols down

to a single wire – funnelled). You associate a distinct identifier with each source. *Demultiplexing* redirects the contents of one channel towards several destinations (mail in the mailbox for multiple roommates) – it uses the identifiers from the multiplexing stage to redirect.

For the various layer components, the data has different names (see above), and the devices and addressing of these layers differ. PDUs are addressed by *port* and are typically associated with *gateways* or *firewalls*. Segments are addressed by *protocol* and are associated with *transport* devices. Packets are addressed by *IP address* and are associated with routers and layer-3 switches. Frames are addressed by *MAC addresses* and are associated with *bridges* or layer-2 switches. Finally, *bits* at the physical level are addressed with *I/O cards* and associated with *repeaters*. Note this list of devices is purely academic, and those above will have the functionality of those below.

NOTE Ports are 32-bit and from 0 to 1023 are reserved for protocols and defined by standards. From 1024 to 49151, they can be registered. Finally, from 49152 to 65535, they are dynamic or private ports.

See the slide on addressing for more information on those. MAC addresses are split in half into manufacturer prefixes and an ID.

Network Address Translation (NAT) provides a way to have one host be a proxy for the network traffic of several other computers. Allows private networks to use the reserved IPv4 addresses while still giving computers access to the internet. The devices that do this are known as *NAT box* with a private address for the private network and a public address for the rest of the internet. In a sense, multiplexing is happening here through this box.

Subnets, or subnetworks, are logical divisions of an IP network into smaller networks. They are typically used to isolate traffic among computers and to secure small networks. They are exposed in the interpretation of an IP address at the network layer: a 32-bit address will contain *subnet bits* and *host bits*. The division between the subnet bits and the host bits is defined by a *subnet mask*. The division in the IP address is not fixed.

Early on in the network world: it was identified subnets were needed, but people did not believe they needed very many. At first, there were 5 classes: A, B, C, D, E. This is called *classful networking*. Most of the internet is designed in unicast (one-to-one communication), but there is another part known as broadcast, and finally another part deals with multicast. At first, the first byte contained a few numbers at the beginning to categorize it in a class (0 to class A, 10 to class B, 110 to class C, 1110 to class D). However, IP addresses began to get pulled back. The boundaries were convenient at first, but they no longer fit all of the sizes of networks we can deal with.

NOTE Broadcasts stay inside a subnet.

Classless inter-domain routing (CIDR) works similarly to subnet masks. NetworkID/SubnetLength defines the the number of bits reserved for a subnet. By convention, host bits are set to 0 in the NetworkID.

Network Structure

The hierarchy of routers, switches, and gateways comprise the **network core**. The **network edge** or "last mile" is the connection from your ISP to your home. Includes dial-up, DSL dedicated lines, cable modems, fibre optics, GSM, WiMax, etc.

Network Structure continued on next page...

NOTE Note that for any wire-based technology, running a current through them induces a magnetic field. They could potentially interfere with each other creating cross-talk. Different technologies coordinate different configurations of wire within the cord.

Home networks enter via a DSL or cable modem. Will comprise or contain a firewall, NAT server, Dynamic Host Configuration Protocol (DHCP) servers, wireless access points (WAPs). Furthermore, there will be local connections via ethernet (IEEE 802.3) and wireless (802.11).

NOTE More notes are located in the scribbler here on NETWORK ARCHITECTURES. From lab on Wednesday, January 16, 2012, it was seen that most packets are either extremely small (50 bytes) or very large (1400 bytes). This is because of differing protocols. TCP, for instance, has this sort of distribution. But, UDP, on the other hand which is used by traffic where it does not matter as much if the traffic goes to the other end (streaming) has very small packets. Furthermore, you can use a program like *traceroute* in order to see how your network works on the premise of time to live (TTL).

NOTE Assignment 1 due next week on Wednesday.

There are three different **network architecture**: (1) client-server, (2) peer-to-peer, and (3) hybrid (see paper notes for more information). Another "architecture" is **cloud computing**. In the cloud, an entire program is run or serviced on someone else's hardware. There are three types: infrastructure as a service (IAAS), platform as a service (PAAS), and software as a service (SAAS). Each of these encompasses the scope of the previous types and are offering to a client with underlying hardware and host OS (with a layer of virtualization).

1. **IaaS**: Computing and storage resources.
2. **PaaS**: Above, and application development environment.
3. **SaaS**: Above, and applications served over the internet.

Definitions vary for cloud computing: see Foster et al. (2008), Plummer et al. (2008), and Mell and Grance (2010). The way we use the cloud today, all three of these definitions are correct. If you look at these and see what is common:

- **virtualization**: hardware can host many independent simulated servers – simulating OS on hardware; can accommodate elasticity or **load sharing** (e.g. multiple Google servers answering queries),
- **multi-tenancy**: multiple clients can occupy the same physical hardware,
- **security**: clients are protected from each other; data is secure,
- **elasticity**: resources can be added or removed in real-time often at the request of the client and without intervention from the service provider,
- **availability**: the service provider gives performance and QoS guarantees,
- **reliability**: failure of any piece still allows services to be offered,
- **agility**: resource allocations can adapt dynamically,
- **pay-as-you-go**: client just pays for resources used.

Something else to note is that clients often use the same way to get to the service no matter how or where the service is deployed.

Introduction continued on next page...

Circuit and Packet Switching

What are different models for sending messages across a network? **Circuit switching** dedicates resources along a path between two hosts for their communication. This comes from telephones. The path is established in the network where all the packets from one computer to another will follow the same path each time. This guarantees quality of service (QoS), switch capacity, and bandwidth. However, it requires set-up (this is the *call set-up phase*) in advance and a commitment from all intermediate network devices until communication is ceased.

There are typically two different methods for establishing circuit switching (these cannot dynamically change); two ways to dedicate transmission resources (neither will give you more data):

- **Frequency division multiplexing** (FDM): split the transmission frequency range (bandwidth) across a fixed number of circuits for as long as they are needed (think of a cable modem – little traffic but always needs to be there).
- **Time division multiplexing** (TDM): give each circuit a periodic interval of time in which the circuit has access to the full bandwidth of the transmission medium (bursty).

The advantages of circuit switching is that it is simple to operate once set-up is done, smaller addressing once the circuit is established, and guaranteed performance (great for applications with a steady stream of information). The disadvantages are that it has set-up time and complexity, a limited number of circuits through one device, and resources are idle when no communication is needed on the circuit.

Packet switching ("statistical multiplexing") is a little more *opportunistic*; paths are up to the devices and the order may not arrive as it came in to the target. Packets are opportunistically transmitted when the medium is available. For this to work, the recipient has to be defined in the packet. The context has to be established. Also, order has to be reclaimed when the packets arrive.

Resources in the core are not held for any particular transmissions in this method. Each packet is self-contained and each packet is routed independently of the other packets in the same conversation. Different choices of scheduling can be applied to the incoming packages – no default guarantee of when a particular packet may be transmitted.

The advantages of this method is that there is no set-up time, any number of incoming hosts are served, and there is an efficient use of transmission resources – this is great for bursty traffic. The disadvantages are that each packet must be self-contained (overhead and addressing must be contained in each packet so devices know where to direct the packet) and there are no performance guarantees (time it will take to get to the recipient); it is subject to congestion, out-of-order delivery, and **packet loss**.

This introduces us to three new concepts which come about from this model: (1) store-and-forward, (2) packet loss, and (3) delay.

Store-and-forward underlies a great deal of packet switching; it is the idea that a complete packet must be received (and *complete*) before we can begin to transmit it. Therefore, each network interface needs some degree of *storage* (a **receive buffer**) in order to carry this procedure out. This is contrast to circuit switching where the path is guaranteed: nothing will be lost and it will not be congested; bits will be sent along in sequence. This technique is also needed in order to bridge between *mediums with differing transmission rates.*

Note, though, that an outgoing network interface may be handling packets from many incoming network interfaces. A network device may need to store many outgoing packets for each interface (a **transmit queue**). Ultimately, this approach injects a **delay** in transmitting data through a network device. The first bit of a package must wait for a last bit of the message to arrive before it gets to start being transmitted.

NOTE As of Wednesday, our class will be in Henry Hicks, room 217.

Packet loss is the idea that each network device has *finite memory*. It can only hold a finite number of messages in its queues. **Network congestion** happens when queue sizes cause a deterioration in network quality of service. Packet loss occurs when queues reach their maximum capacity and messages continue to arrive. *Which messages should be discarded*? There is no fixed idea; it is merely a policy of the network provider.

- Most recent arrival?
- Packets that have not transmitted for a long while (latest arrivals)?
- Most hops made?
- Least hops made?
- Less-important packets?
- Smaller packets?
- Bigger packets?
- Packets destined to a slower medium?

Delay in packet switching come from processing delay (d_{proc}), queueing delay (d_{queue} — see above), transmission delay (d_{trans}), and propagation delay (d_{prop}). This will lead us into **queueing theory**.

1. **Processing delay** involves determining the outgoing interface; it depends on node performance, protocols being interpreted, ... (time — less than 1-9 milliseconds).
2. **Queueing delay** involves the time waiting for the interface to become available for transmission; depends on the congestion at the device (time — varies).
3. **Transmission delay** involves the time to put the bits of the packet onto the wire; with L bits and a **transmission rate** of R bits/second, it takes L/R seconds (we refer to this typically as a function of **bandwidth**).
4. **Propagation delay** is the time for the bits to travel from one end of a wire to the next; speed is about 2×10^8 m/s (copper); time is d/s seconds.

Together: $Delay = d_{proc} + d_{queue} + d_{trans} + d_{prop}$. Less hops will typically result in less delay (the number of times you transmit, queue, process, etc.).

NOTE A common misconception is between 1 MB and 1Mbps; 1 MB = 1×2^{10} bytes = 1024 bytes. 1 Mbps = 1000000 bits/sec = 1×10^6 bits/sec.

You may wonder if there is any overlap between transmission and propagation? By the time the last bit of a packet is on the wire, the first bit has almost arrived to the next device; however, because we are using store-and-forward, there is no problem here – they are treated separately.

Glossary

- IETF — Internet Engineering Task Force
- IEEE — Institute of Electrical and Electronics Engineers

Glossary continued on next page...

- W3C — World Wide Web Consortium
- ISO — International Standards Organization
- ITU — International Telecommunication Union
- NAT — Network Address Translation
- WAN — Wide Area Network
- MAN — Metropolitan Area Network
- CAN — Campus/Corporate Area Network
- LAN — Local Area Network
- VLAN — Virtual Local Area Network – A local area network that may not be in physical proximity.
- PAN — Personal Area Network – Bluetooth-range networks.
- BAN — Body Area Network – Wearable computing.
- NFC — Near Field Communication – More of a technology than a network; communication between a network of two.

Queueing Theory

Overview

Queueing Theory is a mathematical modelling of how queues behave; addresses the size of a queue and the waiting time in a queue. Modelling is an approximation: it is often done in conjunction with simulation so small changes can be made and tested on-demand. This will allow us to model part of the network: think of a supermarket with queues/lines at checkouts.

Parameters of the model(s) include:

- interarrival-time probability density function,
- service-time probability density function,
- number of servers,
- queueing discipline (**FIFO** or priority),
- and buffer space (**infinite** or finite)

The **interarrival-time probability density function** is a graphing of time spent between each packet coming in and an analysis of this distribution. How fast are packets coming in, and can this be characterized? The **service-time probability density function**, on the other hand, looks at how long it takes for messages to be sent out — how long does it take to service? This can also be characterized and looked upon over an average.

Queueing theory models are designated in the format $A/B/m$ where A = interarrival-time probability density function, B = service-time probability density function, m = number of servers. Probability density functions include:

- M (Markov) – exponential probability density
- D (Deterministic) – all values are the same; uniform

Overview continued on next page...

- G (General) – arbitrary probability density

We will concentrate on the M/M/1 model. Arrival rate and service rate will be distributed and governed by an exponential density function (Markov).

Assumptions in the model are numerous:

1. Probability that n packets arrive during an interval of length t is a Poisson distribution: $P_n(t) = ((\lambda t)^n/n!) \times e^{-\lambda t}$ where λ is the mean arrival rate. The distribution will allow us to not worry about the history and minute boundaries.
2. Processing of individual packets is independent of the other packets.
3. Processing time follows an exponential distribution. This assumption is hard to defend, but very long service times are rare; this assumption is only an approximation.

Distributions

As above, **poisson arrivals** imply an exponential interarrival-time distribution. The probability of a packet arriving between time t and time $t + \Delta t$ is:

$$P_0(t)P_1(t)(\Delta t) = e^{-\lambda t} \times \lambda \Delta t e^{-\lambda \Delta t}$$

In the limit as Δt goes to 0, we get:

$$\lambda e^{-\lambda t} dt$$

The consequence of this is that the arrival time of the next packet does *not* depend on how long we have been waiting.

Note that mean arrival rate is λ packets/second and the mean interarrival time is $1/\lambda$ seconds/packet. Similarly, the mean service rate is μ packets/second and the mean service time is $1/\mu$ seconds/packet. Let p_k be the **equilibrium probability** that there are k packets in the system – queued and being serviced. An oscillation between two different states and in one stable spot. A queue time of 0 in an equilibrium would be most ideal: no messages are waiting.

Equilibrium

We can establish a **birth-death system** to model *non-simultaneous events* where edges are labelled with the probability of taking that transition (e.g. another item enters the queue or an item leaves the queue). States are labelled with *queue size* (e.g. number of packets in queue) up to $k + 1$. At each state, there is a probability that the queue has that number of given elements p.

Note that you cannot go from one state to one that is further than an adjacent state away. Because we are looking at an M/M/1 model, you can only service or take in 1 item at a time.

How often are the transitions occurring? We look at the probability of taking that transition – we have a mean service rate of μ and a mean arrival rate of λ. The number of transitions at any event are λp_i or μp_i.

If we are at an **equilibrium**, the transition probabilities are *balanced* and equal:

$$\lambda p_i = \mu p_i$$

Equilibrium continued on next page...

From here, we can solve for *individual values*:

$$p_k = \rho^k p_0$$

ρ here is defined as the **traffic intensity** (how busy are we); we need $\rho < 1$ for an equilibrium.

$$\rho = \lambda/\mu$$

For example:

$$p_2 = \lambda/\mu \times p_1 = (\lambda/\mu)^2 p_0$$

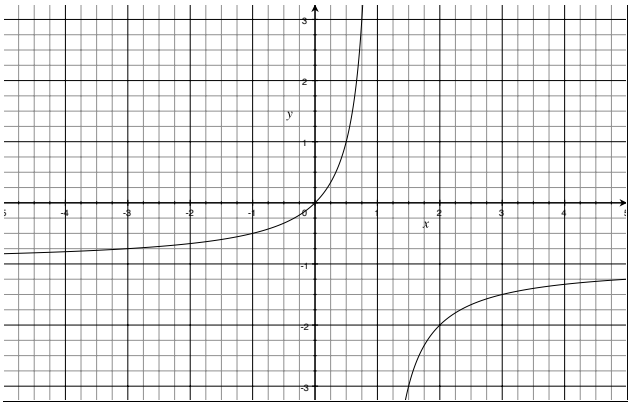
Note that *all probabilities must sum to* 1: $p_0 \Sigma_{k=0}^{\infty} \rho^k = 1$. This produces a geometric series:

$$p_0(1/(1-\rho)) = 1$$

Solving for p_0, we find that:

$$p_0 = 1 - \rho$$
$$p_k = (1-\rho)\rho^k$$

We can then define that $N = \rho/(1-\rho)$ (see slides for derivation) where N is the number of packets (or items) in the system. This expected value of that finite state machine allows us to present this value. See Figure 1. There is a maximum amount of memory for any given device (for number of packets) – as ρ approaches 1, the distance between the curve and 1 is an indication of packets being dropped past the maximum memory. Therefore, the traffic intensity grows *asymptotically*.



Little's Result involves T, the expected waiting and service time, N, the expected number of packets in the system (queue & service), and λ, the packet arrival rate. A specific packet as it arrives is chosen. As that same packet ends, we expect N packets to be in the system, and therefore:

$$N = \lambda T$$

Combining this result with the expected size of the system:

$$T = 1/(\mu - \lambda)$$

We can look at two systems, for example (see Table 1). If one cares about the resources (such as memory), then the ratio of arrivals and processing; if the time is what is cared about, the difference between those is what is important.

System	μ	λ	ρ	T	N
1	$1000p/sec$	$800p/sec$	0.8	$1/200sec/p$	4
2	$2000p/sec$	$1800p/sec$	0.9	$1/200sec/p$	9

Table 1: An example of two systems.

The consequence of the Poisson distribution is that we do not have to worry about the history when we do an analysis. The assumptions of the model mean we do not have to care about this.

Example. Consider a link on a router with 4000 packets arriving every second, a capacity to process 5000 packets per second, an average packet size of 550 bytes, and a router memory size of 1MB for the link. Is the memory for the link sized okay? This means there is a $\rho = 4/5$ and on average, the space needed for packets would be $4 \times 550 = 2200$ bytes. This means the chip can handle a great deal more packets.

For the same router link, at an arrival rate of 4996.7 we can expect to have the buffer filled to 80% capacity in steady state. This is because 1525.2 average sized packets can be held in 80% of a 1MB capacity. Unfortunately, at this time, a little bit more will create huge differences for memory consumption. So, this is not a stable situation to be in.

Throughput

Throughput is the *rate at which bits are delivered from end-to-end* between two computers across a network. It is bounded above by the smallest bandwidth on the path between the two computers. This will involve an entire network of devices. There are two measures of this:

1. *Instantaneous*: throughput at a given moment in time.
2. *Average*: throughput across a period of time.

And there are two possible interpretations:

1. *Bits that are communicated.* Not just messages put into the system.
2. *Complete messages that are communicated.* How many made it to the end.

We will focus on the latter.

Pairs of communicating computers may share a common link. The throughput for each pair of communicating computers is affected by the other traffic on the shared link. The throughput between A and B is the lesser of:

- throughput of A,
- throughput of B,
- throughput of the shared link(s).

Throughput continued on next page...

What could prevent the throughput from achieving the maximum bandwidth? If one pair is trying to use it at full capacity, no one will be able to. If the resources are dedicated by circuit switching, this would not be a problem. If no single pair is able to use it at full capacity, then the problem may be the connections are all backing off and not using the resources in fear of greater capacity. And finally, whenever there are collisions in a network, they will back off and try again later — this is wasted time and decreases the performance. Watch for *bottlenecks*.

How secure are networks? The original vision was that a connection involved a set of mutually trusted computers connected transparently by a network. There was little thought to security at this point:

- unencrypted packets,
- cooperation for routing packets,
- ease of becoming a peer in most protocols (routing, e-mail, domain name translation, file sharing, ...),
- no built-in authentication,
- and little structure to limit resource hogging.

Network designers retrofit security to the protocol stack.

Socket Programming

Goal & API

The goal of this section is to *learn how to build client/server applications that communicate using sockets*. The **Socket API** was introduced in BSD4.1 UNIX (1981). It was explicitly created, used, and released by applications. This also brought forward the idea of the **client/server paradigm** and relies on this foundation. It creates an abstraction of how networks behave.

There are two types of transport services via socket API, two types of connections. These involve what sort of communication will occur at a given port. A given computer program will contain, inside it, the application layer logic. It connects directly to the transport layer and it is there that these reside:

1. unreliable datagram (UDP),
2. reliable, byte stream-oriented (TCP).

While the transport layer is reliable, there is no guarantee. TCP ensures this while UDP is created upon the premise that this is not necessary.

Sockets are created at physical ports in the appropriate layer, and is associated with that port and the communication occurring through it. A socket is a door between an application process and end-to-end-transport protocol (UDP or TCP).

> **NOTE** UDP is used when the time is important. Data wants to be kept up-to-date and live; any bytes that are lost are not important, but it is more important to stay along the same frame of reference of a given process. Think streaming and broadcasting.

Different programming languages employ this in different ways and, by design, work under a client/server paradigm: one far-end listening for connections and one application going and requesting information.

Accepting connections from multiple clients can be done either using **bitmasks** (see select code) or using **threads** (e.g. run multiple loops in parallel using the same memory space).

Socket Programming using TCP

The TCP service ensures and guarantees a reliable transfer of *bytes* from one process to another. The process is controlled by an application developer and between it and the operating system is a socket. The TCP with buffers and variables is controlled by the operating system. Between a process and another is the Internet or a network.

First, we must discuss *stream jargon*. TCP does not look at the world as messages; it looks at it as a continuous sequence of bytes (which we can later separate out as messages). A **stream** is a sequence of characters that flow into or out of a process. An **input stream** is attached to some input source for the process (e.g. keyboard or socket), and the **output stream** is attached to an output source (e.g. monitor or socket).

You do not typically open a file to read and write to it at the same time. File I/O is typically unidirectional. However, in this case, the network communication is **bidirectional**. There is an input buffer and output buffer which keeps data and waits for sending or receiving.

Since we are using a client/server sort of model, we need to figure out which things each side has to do. In this model, the client must contact the server.

- The server process must first be running.
- Server must have created a socket (door) that welcomes the contact.
- When contacted by a client, the server TCP creates a new socket for server process to communicate with the client.
  - This socket creation is initiated by the OS.
  - This allows servers to talk with multiple clients.
  - Source port numbers are used to distinguish clients.

The client contacts the server by:

- Creating a client-local TCP socket.
- Specifying an IP address, port number of server process.
- When the client creates this socket: client TCP establishes connection to server TCP.

From the point of view of the **server**, there are some generic server steps.

1. Create a *socket* for the server, specifying stream data.
2. *Bind* a socket to a port, and optionally, which IP addresses can be accepted.
3. Tell the OS to *listen* for up to X clients in a queue.
4. Wait for a client to arrive and *accept* the client connection. This connection gets it own private client socket.
5. *Read* and *write* (or *receive* and *send* messages) to the client socket, sometimes *selecting* input from more than one socket/stream.
6. *Close* the client socket, but not the server socket.

From the point of view of the **client**, there are some generic client steps.

1. Create a *socket*, specifying either stream data.
2. Translate domain name to an IP address.

Socket Programming using TCP continued on next page...

3. *Connect* to a specific IP address and port number through a port number that is the OS' choice (this information will be sent along to the server).
4. *Read* and *write* (or *receive* and *send* messages) from the socket.

Socket Programming using UDP

UDP differs only in the fact that there is no "connection" between client and server. There is no handshaking, the sender explicitly attaches an IP address and port of the destination to each packet, and the server must extract the IP address and port of sender from the received packet. Furthermore, UDP's transmitted data may be received out of order, or lost.

See slides on server/client programming steps. Generic server steps include:

1. Create a *socket* for the server, specifying datagram data.
2. *Bind* a socket to a port and, optionally, which IP addresses can be accepted.
3. *Read* and *write* (or *receive* and *send* messages) to the client socket, sometimes selecting input from more than one socket/stream.
4. *Close* the client socket (not the server socket).

Generic client steps include:

1. Create a *socket*, specifying datagram data.
2. Translate domain name to IP address.
3. *Connect* to a specific IP address and port number.
4. *Read*/*write* (or receive/send messages) from the socket.
5. *Close* the socket.

Application Layer Protocols

How can you tell when logical pieces of information start or end? How do you ensure that the content is understandable? Do you maintain a notion of context and, if yes, then how? How do you allow your protocol to be extensible? How do you deal with having old clients on the Internet or clients that you cannot certify? How do you make the protocol as friendly to the network as possible? How do I handle messages that are out-of-context? How do I keep control of the protocol when the network becomes congested?

HyperText Transport Protocol (HTTP)

Also known as the **HyperText Transport Protocol** — most popular use is to transport HyperText Markup Language (HTML) content for web pages. Transfers files between a server and a client.

A web page consists of objects (image, audio files, HTML file, video, ...). A web page usually acts as a base HTML file that references the contained objects. Web references occur through a **Universal Resource Locator** (URL): `http://www.cs.dal.ca/undergraduate/bcs` where the first part (http) refers to the access method, the next part (www.cs.dal.ca) refers to the server, and the last bit refers to the path to the resource.

The **web client-server model** features clients that request URLs through a web browser and display content to the user. Usually you have one request for each URL. Servers respond to client requests (one response for each request). Finally, the server listens for web requests on port 80 (default; others can be specified with a colon).

HTTP operates over TCP: data is reliably delivered. No state is maintained between the client and the server; therefore, the protocol is **stateless**. "State" would consist of history between the client and the server, past requests, type of client or server being used, number or size of responses, ... Stateless protocols are usually less complex than **stateful protocols**. There is no state to restore if one of the two communicating parties disconnects then reconnects.

HTTP connections to the server are:

- non-persistent,
- or persistent.

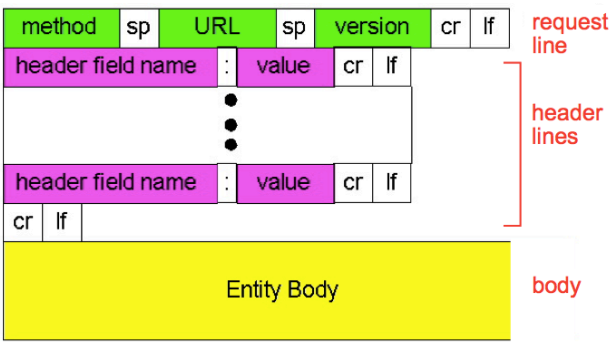
Non-persistent connections have each URL request open a new TCP connection for the object and then closes the connection once the object is received. This is HTTP version 1.0 default. Web pages generally had few objects, networks were slower (so connections stayed open longer), and connections at the server could be scarce. See slide for HTTP 1.0 diagram. Makes it easier for the protocol to work but can create extra web traffic.

Persistent connections have clients that keep a single connection open to the server and can send multiple URL requests through one TCP connection. The connection is closed by the client when it has received all the needed data from the server. This is HTTP version 1.1 default. Web pages often have many objects from the same server and the networks are fast enough to keep connection times short; a single connection reduces set-up time from TCP. See slide for HTTP 1.1 diagram.

NOTE TCP has a **three-way handshake** involved in connection requests in order to acquire data – see diagrams. One way to avoid having to do multiple handshakes in a row (such as in HTTP 1.0), *parallel connections* could be used in order to speed up the process.

NOTE For subnets, use the subnet slash notation, or give the subnet mask (e.g. 255.255.255.192). Example: given 10.0.0.64/26, the number of addresses available would be $2^{32-26} = 64$.

Version 1.0 HTTP requests include GET, POST, HEAD. 1.1 adds PUT (store files), DELETE (delete them), TRACE, OPTIONS, and CONNECT. HTTP responses are comprised of numeric and textual information. Categories include: 1xx (informational), 2xx (success), 3xx (redirection), 4xx (client error), and 5xx (server error). These categories allow for **graceful degradation**: older browsers can still understand newer codes.



HTTP requests messages have a **general format**: a **request line**, **header lines**, and the **body**. See slide. The header has a minimum of one line (request line) – the method, object (to do it on; webpage), and the version number of the protocol being used. Line feed (lf) and carriage returns (cr) act as the last two bytes. A blank line separates header information from a body (to get a webpage, you do not need a body).

HTTP response messages are very similar. They differ only in the request line where the method is replaced by a version, the URL replaced by a **status code**, and the version replaced by a **status message**.

NOTE HEAD only retrieves the header information and not the entity body of a response message; it is useful if the body information is already possessed. TRACE is used for debugging – see if connection is working okay. OPTIONS allows for exploring what else the server can do for you.

Web page parameters have parameters as name=value pairs. GET parameters are included in the URL and exposed to the user (separated by the base by a ?, individual parameters separated by &). POST are included in the body of the HTTP request; unseen by the user and not stored with a bookmark.

HTTP **headers** allows for extensibility of the protocol. Servers interpret the headers that you recognize and ignore other headers it does not need or recognize. Avoid creating headers that depend on one another (or that modify another header).

Two header lines that are added to help the client and server manage *state* are:

```
set-cookie: ...
cookie: ...
```

State management requires four components:

1. `set-cookie` header in an HTTP response,
2. `cookie` header in subsequent HTTP requests,
3. cookie file stored at the client,
4. database at the server to track the relevant state; the cookie will typically contain some unique ID for the client, generated by the server — the variables and computations will be related to this key.

In addition to a cookie, there will be usually a set of check digits in order to ensure consistency and message integrity; e.g. a hash function — prevents exploitation.

A **cookie** can allow *authorization*, *shopping carts*, *recommendations*, and *user session state* and can have an expiry date specified. They are usually invisible to the user; it can track you as you go between sites (websites include a reference to a non-visible object hosted by a tracking company; e.g. a 1-pixel image) and information can be supplied about your history at a website without your knowledge.

A potential problem that can arise is in GET parameters. Including sensitive information in GET parameters or cookies opens that information to changes from the user; the user can change the URL or the user can edit the cookie file. This allows the user to send invalid information to your server or to scan for valid information.

Caching refers to the storing of common content to avoid redundant network traffic. This can be done at the client or within the network; it is not uncommon for ISPs. The benefits of this is that it reduces response times for client requests and reduces traffic on an institution's internet connection. However, the costs is that it could deliver out-of-date content (if expiry time is not managed correctly) and must manage which content to keep.

Proxy servers are servers within a network that allow HTTP clients to contact them through an explicit request, can be configured in the web browser, or re-routes requests by the network forcibly (e.g. an institution or company). The proxy obtains the object from the origin server and serves it in future requests (as it has its own cache).

Conditional GET are GET requests that ask to complete the request only if local information is out-of-date – a header line is added to the HTTP request.

```
if-modified-since: <date>
```

A response 200 OK will be received if content is out-of-date (content supplied), or 304 (not modified) if the local copy is current (no content supplied). This is used by clients and web proxies.

In summary, the protocol HTTP has the following features:

- easily readable format (ASCII),
- connection **persistence**,
- separation of headers from **payload** (message body),
- **delimiters** to denote field separation (end of header lines, separation of header title and values),
- extensible headers,
- identification of protocol and protocol version in the header,
- structured **status code**s,
- coordination of client/server to maintain **state** separate from protocol,
- and caching support.

File Transfer Protocol (FTP)

The **File Transfer Protocol** (FTP) is a protocol used to transfer a *file* between a client and a server. It is a **stateful protocol** — the client and server know information about the interaction (e.g. current working directory, binary or ASCII transfer, ...). A much older protocol for transferring files but there are a few good take-aways considering how simple it is.

The most interesting part of the protocol is that the protocol employs **out-of-band control**: two ports are allocated (20 and 21) where the server lists on port 21 for connections (maintaining a persistent connection) and responds with data on port 20 with a non-persistent connection. One line of communication (a high-priority channel) is already maintained, therefore, in order to track commands and communication even while data is being transferred (the bandwidth is separate and commands are outside of that bandwidth). With HTTP, this would have been called **in-band control**: one socket is used.

FTP has an **active mode** and **passive mode**. Its active mode is where the server initiates the data connection to the client. This is normally how things are done. The passive mode is when the client initiates the data connection to the server and lets you theoretically connect two servers together to exchange files while being managed by a third host.

See slide on commands, responses.

The main features of FTP is that:

- has separate **data and control channels**,
- has specifications of types of data,

File Transfer Protocol (FTP) continued on next page...

- and has structured status codes.

Historically, FTP allowed *anonymous login* with your e-mail address as a courtesy password — could be used as a launch-point for an attack on a system. This is often disabled in networks today in favour of other file transfer alternatives. Earlier, if things were not mailed to you, files were received in this manner.

E-mail

E-mail consists of:

- mail servers,
- user agents (clients – program used).

Mail servers typically talk to other mail servers or to a client (user agent). Core mail transfer occurs through **Simple Mail Transfer Protocol** (SMTP) – port 25. This is employed between mail servers, from user agents to mail servers to send e-mail.

A mail server communicates to the client using POP3 or IMAP (see below). Mail management by the user agents and the mail server is done through **Post Office Protocol** (POP3) – port 110 and **Internet Message Access Protocol** (IMAP) – port 143.

Mail encoding is done through **Multipurpose Internet Mail Extensions** (MIME). Earlier, the basic protocol only carried simple text messages. As that functionality became needed (to send binary messages), MIME came about. On the TCP/IP stack, this usually all happens at the application layer (with MIME being particularly on the presentation layer in OSI).

Mail servers employ a peer-to-peer-like architecture among other mail servers. They are always on and ready to trade e-mails with each other. On the other hand, client-server architecture is employed between mail servers and user agents. Each mail server has:

- a mailbox for each user at the server,
- a queue of outgoing messages to send (typically aim to deliver as soon as the message is received from a user agent; queue is needed if destination mail server is not available).

NOTE IMAP has a different view: e-mails are accessed from more than just one device. Most mail servers these days will have an entire directory structure which can be organized on the mail server.

A distinction has to be made between **push** and **pull** here. Applies to many other sort of interactions, not just e-mail. This is a difference of actively and passively communicating information.

A **pull** protocol is one where the recipient of information must ask for the information or updates on it. This is akin to polling for information. Examples of protocols that employ this are: HTTP, POP3, IMAP. Useful when clients are transient.

A **push** protocol is one where the source that has information actively tries to deliver the information to recipients. An example of this is SMTP. It becomes cumbersome on the server but is used when there are stable recipients.

SMTP involves direct transfer of e-mails to a mail server and contains three phases:

1. handshake (greeting) — HELO, EHLO,
2. message transfer (can transfer multiple messages in one connection) — MAIL FROM, RCP TO, DATA,

E-mail continued on next page...

3. and closure — QUIT.

All interaction is in 7-bit ASCII characters. Lines end with a CRLF character combination. It *does not specify the header lines for the message itself.* In the end, the main commands SMTP features are shown above.

The Mail Message Format is defined in RFC 822. It consists of header lines (`name: value`), a blank line, and the message body (ASCII text only). Intermediate SMTP servers can add to the e-mail headers of a mail message, adding meta-data, tracking information, and describing transformations to the data.

Challenges with **SMTP** include:

- no authentication in the basic protocol (delegated to firewalls, secure connections, ...),
- odd HELO/EHLO introduction (not a convenient way to migrate a new version),
- no native support to send binary data (relies on an additional protocol).

MIME allows for multiple parts in a message; allows one message body to contain several pieces. It is defined by RFCs 2045, 2046, 2047, and 2049. Each piece is separated from another by a boundary delimiter: preceded by `--` at the start of the line. Delimiter only valid within one message. Each piece contains its own specification of `content-type` and `content-transfer-encoding`. Examples: `text/plain; multipart/alternative; image/jpeg; video/mp4;`. The alternative one is analogues to an `alt` attribute for an image in HTML — gives an alternate form.

```
Content-Type: multipart/mixed; boundary=XXX
```

Above is a main SMTP message body header found at the top of a message — defines boundaries. The MIME headers indicates type and subtype, `XXX` is the MIME delimiter value within the message.

POP3 assumes *single access* to the mailbox. It was originally intended to let users download their e-mail (which deleted it from the mail server). Current updates allow the user to delay the deletion. It maintains state within a session but maintains no state across sessions. Furthermore, it is intended for short connections to the web server. Client commands (RFC 1939) include: `user, pass or apop` (for authentication) and `stat, list, retr, dele, noop, rset`.

IMAP4 assumes that a user may be connected from *multiple hosts.* Intended for e-mail messages to remain stored on the mail server. It also allows more than one mailbox (folder) per user. It maintains user state (folder organization) across sessions and expects long-lived connections. Client commands (RFC 2060) include: `authenticate` or `login` (to authenticate) and `capability, noop, select, examine, create, delete, rename, subscribe, unsubscribe, list, lsub, status, append, check, close, expunge, search, fetch, store, copy, uid`.

NOTE A question on the final exam last year: you are an ISP provider... which of the two protocols would you prefer to give to your clients and why? POP wins for scrooge ISPs. For the happiest customers, you go towards IMAP – more complicated to manage, more space.

Where does *spam detection* or *management* fit in? We figure it is an issue, most of us do not want to deal with it. According to statistics, in 2004-2005, 90% of incoming e-mail was bogus (Dalhousie ITS). 34% of messages were labelled as phishing, 1% were regarded as viruses, and 4% were missed (as spam).

Site-level filters include: **white list**, **black list**, and a **grey list** (do not trust completely, but do not distrust). Content filters look for markers that a message could be spam, accumulate the markers as evidence for or against being spam, and provide a **spam score** and then filter based on that score. Game of cat and mouse: no definitive rule for spam filter. Can also be adjusted for how aggressive it can be.

E-mail continued on next page...

NOTE MX usually means mail exchange (mail server) in addresses, AV means anti-virus, and SM usually means secure mail.

The overall take-away from e-mail is that it is:

- a peer-to-peer interaction,
- has separate protocols for "last mile" (different roles),
- nesting of protocols in the TCP/IP stack (MIME in SMTP messages),
- push *v.* pull protocols,
- extensible headers (like HTTP),
- intermediate nodes adding to the message (tracking path, intermediate processing),
- and an initial handshake.

Domain Name System (DNS)

Domain Name System (DNS) provides a mapping between user-friendly computer names (domain names) and IP addresses. It allows one IP address to have multiple *names* and allows one name to map to multiple IP addresses. Therefore, it allows users to remember a stable name while the responding IP address can change as the network changes.

If you think about it, it is one giant distributed database — many-to-many relationships between names and IP addresses. The content of this conceptual database is managed by a hierarchy of name servers (operating on port 53). The distributed nature of this provides:

- no single point of failure,
- better management of traffic volume,
- shorter propagation times,
- localized maintenance of local data,
- and scalability (adapting *gracefully* – predictable change in performance).

Furthermore, DNS provides **host and mail server aliasing** and **load distribution**.

Domain names have an **organizational hierarchy**. For example, `mail.cs.dal.ca` consists of a **top-level domain** (TLD) `ca`, a second level domain `dal`, and a third level domain `cs`.

The **root DNS** maps all TLDs to their domain servers. There are nominally (in theory) 13 worldwide with servers labelled A through M and some of the root DNS are redundant and geographically distributed. However, they are replicated and cloned in some manner and realistically there are a great number of them. Therefore, other servers can be identified:

- *top-level domain name server*: directs you to authoritative DNSs for their TLDs,
- *authoritative domain name server*: provides definitive mapping of a domain name to an IP address and is maintained by the organization whose domain you are seeking; one of the leaves,
- *local domain name server*: provides non-definitive mappings of domain names to IP addresses, usually provided by an ISP or company as the first query point; can cache results from previous queries.

Domain Name System (DNS) continued on next page...

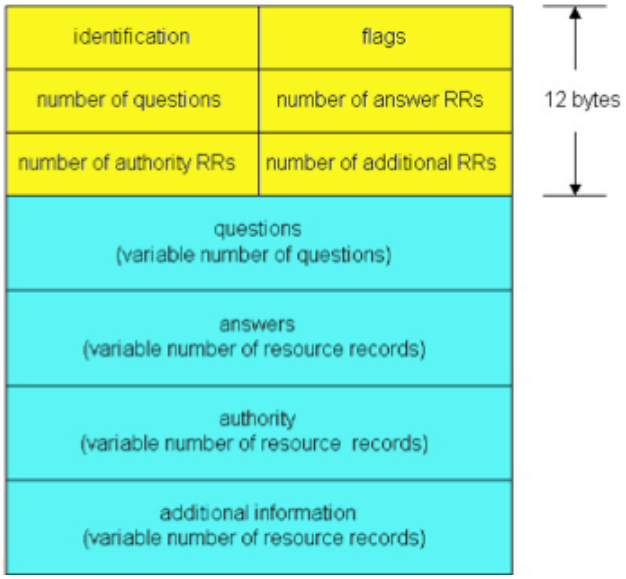
DNS name resolution involves translating a domain name to an IP address. It is a navigation of the DNS hierarchy to reach an authoritative DNS server for the domain name. It operates in one of two modes: **iterative** and **recursive**. Both are "looping" but recursion lets this be done implicitly, but this leverages load back to the root server and prevents caching. Actual operation is usually a mix of these two: hit the root and TLD iteratively and then recursively resolve things at the authoritative DNS.

Caching is what is done at a local DNS (or your host computer) – it caches DNS entries. All local DNSs typically cache TLD DNSs. More opportunities are available to local DNSs to cache with iterative queries. Like any other caching, this has to be kept current: cache entries can periodically expire.

NOTE Subdomains are merely smaller subsets of domains.

DNS records are typically called **resource records** (RRs). Typical content is: `(name, value, type, ttl)`. Most common types of RRs are:

- A — address, maps a domain name to an IP address (*core*),
- CNAME — canonical name (most informative), defines an alias,
- MX — mail record, defines where mail goes for the domain,
- NS — name server, defines an authoritative DNS for a domain,
- PTR — pointer, a generic redirection to another host.



The **DNS protocol** consists of **query** and **reply** messages, both with the same *message format*. Can work over UDP (identification number). The first 12 bytes remain fixed at that amount; the size of the rest of the message is defined in here.

The resource record format is a series of fields that are fixed-length except for the name and rdata. Name is always a domain name (replacing . with length of next string, a binary number, and end of name with a length of 0 – e.g. cs.dal.ca becomes 2cs3dal2ca0). rdata depends on the type and class of the record. Requires one to follow through the whole message to get to the end.

NOTE `nslookup` will query for this sort of information about a domain name.

The take-away from DNS is as follows:

- use of hierarchies (gives scalability; delegation of resolution, leverage/format information in domain name),
- importance of caching (domain names cached),
- iteration *v.* recursion,
- fixed-size header at the start (first protocol we've encountered; usually happens at network layer and below),
- accommodating variable-size blocks,
- use of the infrastructure to help with load balancing and provide fault tolerance (mutliple resolutions for a domain name or for name servers),
- returning different information depending on function (A *v.* MX records),
- providing aliases

See slides on **Application Layer Protocol Issues**.