CS 4516 Advanced Computer Networks Notes

Christopher Myers

June 13, 2020

Contents

1 Internet design principles

1.1 How to design complex systems

Sysstems of networked computers (like the internet) are complex objects too complex to fully grasp all at once. A general principle of design is "Modularity guided by abstraction is the way to get things done" — in other words, components of a system should provide a clean interface that hides away the details of their implementation, the aim being to make systems more modular and thus easier to work with and design.

1.2 What is a network?

A network is a group of nodes (usually computing devices) connected in various ways by a link such that they can exchange data. Links are components that are used to exchange data, like a wireless connection or an ethernet cable. Complexity in networks stemps from making effective use of physical mediums, figuring out how to route data to its destination, ensuring that communication can continue even in case of select failures, and multiplexing communication flows on the same links

Networks break this complexity by first identifying a set of core tasks that the network infrastructure must perform — e.g. routing traffic. These individual tasks are then broken down into protocols, a stack of which is used to build effective networks. The modern protocol stack has several layers:

1. **Physical:** The physical link used — copper, fiber, radio, etc.
2. **Link:** How data is sent across individual links — 802.11 wireless, Ethernet, etc.
3. **Network:** The fundamental structure of how data packets are sent — IPv4, IPv6
4. **Transport:** How devices ensure data arrives in the right way with the right characteristics (speed, reliability, etc) — TCP, UDP, RDP, etc. Usually considered an abstraction for a data "pipe".
5. **Application:** How user-level applications themselves communicate — HTTP, SMTP, etc.

1.3 End-to-end principle

One architectural principle of network design is the end-to-end principle: if some function is of interest to endpoints in some network, that function should be implemented at the endpoints *only*. So, don't duplicate functionality at multiple levels if you have to do it at the top anyways.

There are two main reasons for this. One, duplicate functionality is costly, e.g. reliable transport introduces delays and retransmissions. Also, functionality applied at a low level is applied to *all* applications, which may not be desirable. For instance, TCP might be a very reliable protocol, but it's nowhere near as fast as UDP. Since that part is of interest to end applications, it should be those that worry about it, not other parts of the network, since

if the network imposed TCP-like reliability on all traffic, some applications might perform slower than acceptable.

Note that this isn't necessarily a hard rule; whether or not to obey it always depends on the situation. For instance, implementing network reliability is often a good idea, although technically in some cases the end-to-end principle might advise against it. In such cases it's a good idea to make the specific protocol used decidable by the end application, in much the same way that applications can choose between using TCP and UDP.

Example: End-to-end (E2E) encryption End-to-end encryption is communication where the data is encrypted between each endpoint at an application level. IPsec is technically not E2E encryption since data is decrypted at an IP endpoint and not an application, so a better example is TLS (HTTPS). E2E encryption guarantees that information can't be viewed by third parties while still in the pipe[1], but it may involve additional complexity for applications that implement it.

Often the location of an endpoint needs further definition. For example, consider a simple VOIP system of two connected phones with users listening to each end. By the end-to-end principle, error correction should be on the user's part — that is, if the connection is poor, you should expect your users to just muddle through it. However, this does not imply that you should always let the user take care of the problem (especially if you care about user experience), so more advanced VOIP networks might sacrifice some speed to improve voice quality in the name of better user experience.

The end-to-end design principle isn't just for networks either, and can find usage in distributed system design or other complex design problems. For instance, in CISC vs. RISC CPU architecture philosophy, CISC machines provide complex instructions as processor instructions, while RISC machines provide basic operations only, with the expectation of implementing advanced functions in software. The same principle can be applied to OS design (microkernels vs. monolithic kernels), so perhaps a better formulation of the E2E principle is that functionality should be implemented at the level at which the most information to deal with it is available (paraphrasing from lecture). This should be viewed as a heuristic, **not** a law — it fosters simplicity and elegance of design, but complex systems aren't always remotely simple or elegant.

1.4 Design Rationale

The early internet was built out of a combination of military and acadmic work with the goal of creating reliable computer networks. Commercial use began in the late 80's and early 90's, and NSF stewardship ending in 1995 gave impulse in the transformatoin of the network from a research network to a public and commercial tool. The modern internet resulted in browsers that turned from utilities that display hypertext to full-blown application platforms, with many websites as client-server applications in browsers instead of displaying native UIs. The mobile revolution, starting in the early 2010s, meant that many network

[1]This problem is what Google was caught by with the NSA. The NSA was able to view Google data once it entered Google's networks because Google decrypted traffic on entry to its network, not on arrival to the final destination server.

applications don't run in browsers anymore. Modern cloud computing contributes heavily to the centralization of content in large datacenters, compared with the early internet with many smaller individual servers.

1.4.1 Basic design aspects

The internet uses packet switching as opposed to circuit switching. Circuit switching involves data between two nodes following a static path that must be reserved and set up prior to communication. Concurrent communication between different host pairs must use different paths. This has the benefit of performance and guaranteed performance, but multiplexing communications across a circuit switched network is challenging, and ultimately circuit switching can be more resource intensive.

Packet switching intsead relies on breaking data into packets which are forwarded across links between nodes. Communication from different hosts can be multiplexed by just alternating packets from different sources. Most modern networks are based on packet switching, although vrtual circuits are sometimes used. A concept important to this is store-and-forward, which means that routers on the intenret rceieve and store (in memory) an entire packet, before sending it on the next link. This means that packtes can be processed before being forwarded on, which is important for de/fragmentation, NAT, error checking, or even network analysis & blocking.

2 Network layer

2.1 IPv4

IPv4 was proposed by RFC 791 in September 1981, deployed in 1983, and is still used to route most traffic today. The most pressing and valid reason to change it is address exhaustion — IPv4 only uses 32 bit addresses, or $2^{32} \approx 4$ billion addresses, which seemed like enough at the time. In the initial design, every valid address was expected to be globally routable, with every device on the network expected to have a public IP. Initially, addresses were partitioned with 8 bits for the network and 24 bits for the host address, which was *extremely* wasteful. In fact, this allows for only 256 individual networks with approximately 16 million hosts each.

To solve this, the classful routing system was developed. Class A networks follow the 8+24 bit scheme with the most significant bit as 0, for 128 class A networks. Class B is 16+16 with MSBs 10 Class C is 24+8 with MSBs 110, class D is multicast with MSBs 1110, and class E is reserved with MSBs 1111 (unusable as some network equipment will refuse to route it). This scheme was designed to give better flexibility while keeping hardware simple.

By 1993 even the multi-class system was too wasteful, with a sparsely used address space that was approaching exhaustion. The solution was classless inter-domain routing, or CIDR, introduced in 1993, which uses network masks, etc. for routing. The next emergency measure added was NAT, or network address translation. In practice NAT breaks the internet into a public scope and many smaller, private scopes. Nodes in the public internet must have public IP addresses, and all private networks can assign their own IPs internally.

2.1.1 Network address translation

There are multiple kinds of NAT. The first is "full-cone NAT", which is a 1-1 mapping between an address/port pair an on internal node and an address/port on the NAT gateway. An external host can send packets to the internal address/port by addressing them to the address/port on the NAT gateway. The advantage to this is that hosts on the network are reachable from the public internet.

The next kind is "restricted-cone NAT", in which an external host can only send a packet to an internal host if that external host has previously received a packet from the internal host.

The next kind is port-restricted cone NAT, in which an external host can only send a packet to an internal host if that external host has previously received a packet from the internal host on the same source port.

Finally, the lsat kind is symmetric NAT, where the NAT gateway maps each flow to a specific address/port on the gateway. The difference between this and *-cone NAT is that the latter assigns gateway address/port based on address source/port, while the former assigns gateway address/port by flow. Symmetric NAT is the trickiest to bypass since it makes mapping unpredictable.

When proposed, NAT was controversial, since it breaks the principle of one node to one (or more) public IPs. However, it does decouple the number of internet-connected nodes in an organization from the size of its internet presence. For example, an organization may want all computing nodes to have connectivity, but may only need a few public IPs for its public internet presence, email, etc.

NAT can cause issues on its own though, since sometimes it's useful to allow some internal host addresses/ports to be reachable from outside. There are various techniques for this, such as hole-punching or port forwarding. NAT hole punching involves using an external server to assist two hosts, each behind their own NAT, in which the server provides relevant address/port information to each party to assist them in making direct connections to each other. However, this only works for *-cone NAT since mappings under it remain consistent.

If NAT hole punching doesn't work, port forwarding is the next option, which basically just means configuring an external port on the NAT gateway to always forward traffic to a certain internal host/port. A protocol called uPNP, supported by most home routers, automates the process of setting up port forwarding.

2.2 IPv6

IPv6 was designed to replace IPv4 on in multiple ways, most importantly by having a wider address space. It's designed to ease the transition form IPv4, to provide better security, quality-of-service support, support for host mobility, etc.

The address space was expanded not only because of IPv4 exhaustion, but also because it was predicted that the general-purpose computer market will keep expanding, including devices that were (at the time) referred to as "nomadic personal computing devices" (things like cell phones and laptops). Networked entertainment (like streamed video) was also predicted, and even control over home devices and other hardware was predicted (which is what we tend to call IoT devices today). Designers envisioned a future of much, much more than

one computer per person.

In comparison to IPv4, the IPv6 header is actually much simpler, which fewer fields (7 IPv4 fields were dropped), although the header is two times longer since addresses are four times longer. The vast majority of IPv6 space is reserved for future use, with a further 1/8th reserved for aggregatable global unicast addresses. There are "link local use" addresses that are usable on the local network, but not globally unique — these are designed to be largely autogenerated without need for external information. Typically, they're made using a set series of 10 bits at the beginning (the link-local address format), n bits following, and $118 - n$ bits immediately after that form the interface ID, usually constructed from the level 2 address (like the MAC address of the device).

IPv6 was designed with ease-of-transition in mind. The protocol supports embedded IPv4 addresses, allowing IPv4 connections to be tunneled over IPv6. Second, there are IPv4-compatible IPv6 addresses, which solve the infrastructure issue of routing IPv6 through IPv4-only connections. There's also "teredo tunneling", which is another kind of IPv6-over-IPv4 tunneling, implemented because regular tunneling hides layer 4 and therefore breaks NAT hole punching and related techniques. Terdo works by having a client interact with a Teredo server to find a mapping between local and NAT address/port (a form of hole punching). The server then generates an IPv6 address for the client, which encodes the public address/port used on the NAT gateway. The client then uses a Teredo relay to communicate with other IPv6 users. Of course, there are many more tunneling techniques (6rd, ISATAP, etc.), all designed with different workarounds in mind, exist.

2.3 Routing

Forwarding is the process of using a pre-build forwarding table to decide what port to send packets out on. Routing is the process of establishing those tables to begin with, which is done algorithmically. The two main approaches are distance-vector protocols and link state protocols.

2.3.1 Border Gateway Protocol

BGP is a distance-vector routing protocol for determining routes in such a way that it avoids loops and the bounce effect. BGP defines an "autonomous system" (or an AS) as a set of routers under a single technical administration(a single organization may own multiple ASes). They contain a set of advertised prefixes, and use an interior gateway protocol (IGP) and common metrics to route packets inside themselves. For routing packets outside, they use an exterior gateway protocol. Each routing update carries the entire path as a sequence of ASe. When an AS gets a route, it checks if it's already in the path. If it is, it rejects the route in order to prevent loops; otherwise, it adds itself to the route and possibly advertises it further.

The advantage to to this is that metrics are local – the AS chooses the path and the protocol ensures there are no loops. BGP also uses TCP to connect peers (on port 179), which greatly simplifies the protocol since there is no need for periodic refresh, as routes are valid until withdrawn or the connection is lost.

There are several categories of ASes: stubs, multi-homed, and transit. Stubs are ASs that have only a single connection to one other AS. Multi-homed ASes have multiple connections to other ASes, but they don't allow traffic to pass through them. Finally, transit ASes have both multiple connections *and* allow traffic to pass through them. These all result from changes to BGP policies, which are implemented as configuration changes. Note that BGP is not needed if the network is a stub, if the AS does not providde downstream routing, or if the AS uses a default route.

Paths are selected based on path attributes plus external (policy) information. For example, some attributes might include hop count, presence or absence of a certain AS, path origin, or the dynamics of the relevant link. The AS_PATH attribute, for instance, is a well known mandatory attribute that contains a list of traversed ASes. If the information is forwarded to an internal peer, the attribute is not modified, but it *is* modifed if the routing info is forwarded on to an external peer.

3 Network Security & Monitoring

3.1 Early Attacks

Early attacks in the 90's were system intrusion — cracking passwords, exploiting vulnerabilities, etc. The tool "metasploit" is a good example of this. Later attacks in the late 90's involved distributed denial of service (DDoS) attacks, whcih are attacks that flood a machine with so much traffic (or the right type of it) from multiple sources that the machine is unable to provide service to others. A very old attack is the "ping of death", in which the attacker sends a very large ICMP ping packet with a size of 65kb. The packet would sometimes get fragmented and the target would have to reconstruct it. Because the reconstruction buffer was not designed with large packets in mind, a buffer overflow wsa triggered. Another bug, WinNuke, worked by sending a malicious but correct TCP packet with the "urgent" pointer set, to port 139. Windows 95/NT was unable to process the packet correctly and crashed. Finally, the "smurf attack" was a DDoS amplification attack that worked by broadcasting an echo request to a large group of machines with a spoofed source address; the response would be sent to the spoofed address, which was in reality the address of the attack target.

3.2 Distributed Denial of Service Attacks

The two key ideas to modern DDoS attacks are to compromise a large number of systems over time, and to use them to simultaneously generate large amounts of traffic, using source-spoofing for evasion. Traffic generators could be ICMP or UDP floods, which overwhelm routers that forward packets, or a TCP SYN flood attack. Much of this can be automated — botnets are routinely used to carry out these sorts of attacks.

Modern DDoS attacks frequently stem from botnets, so attribution isn't nearly as important since the attack doesn't stem from the actual attacker (the master of the botnet). New DDoS attacks may leverage smart devices like smart TVs, smart cameras, refrigerators, etc. These devices are typically not designed with security in mind and they frequently lose software support from their manufacturers in short order, so they make prime targets for

those looking to form botnets. For instance, the Mirai botnet is entirely just smart devices with no amplification involved.

3.3 Third Generation Attacks (early 2000's)

Third generation attacks in the early 2000's used self-propagating code (worms) to intrude machines. The idea is to exploit one vulnerability on a large number of deployed systems to rapidly propagate code that runs on the system. The goals of these worms vary — some are used for DDoS attacks (although propagation alone can overwhelm networks), extortions, or other purposes that one might have for a huge swathe of compromised machines. Worm attacks are initiated from a single machine, or using a hitlist.

One example of a worm attack was Code Red I, in July 2001, which used a Microsoft web server vulnerability and ultimately targeted whitehouse.gov. Code Red II used the same vulnerability but had a slightly more sophisticated address probing technique.

3.4 DDoS Mitigation

Large internet companies like Cloudflare, Google, Akamai, etc. offer DDoS mitigation. One solution is reverse proxying, in which traffic towards a website gets distributed to a variety of network locations. Each location applies a number of techniques to filter out likely attack traffic, while only legitimate traffic is redirected to the actual site. Protection can be very effective, but only if you can afford it. For instance, the site krebsonsecurity.com used to enjoy free protection from Akamai, but the company withdrew that service after the level of traffic reached the point of potentially costing the company millions of dollars to cover. The site investigated other options, but it would've cost $150,000 to $200,000 per year. Some DDoS mitigation companies have even been known to shield malicious actors, but other companies offer free mitigation for websites that may be the target of politically motivated attacks.

3.5 Modern Worms

Modern worms are just called "malware". These focus more on financial gain than they do on active propagation, aiming to do things like steal credentials, accounts, bank information, etc. There's a large criminal economy behind them, and phishing campaigns are often used instead of exploits to achieve installation on the target machine. Massive internet-scale disruption is not a goal, in part because it's better to avoid attracting the attention of law enforcement. Rare worm outbreaks still happen and they can be successful, though. For example, the WannaCry and NotPetya worms encrypt files on victim machines, rendering them useless unless the owner pays up (this is why it's called ransomware!). They spread using a set of well-known exploits, e.g. a Windows SMB block.

3.6 Intrusion Detection & Prevention

This field of security is all about detecting intrusion attempts and protecting against them. Intrusion detection can be host-based, detecting the presence of malicious software agents on

networked machines using varoius program analysis techniques. They can also be network based, detecting the presence of communications with malicious purposes.

Network intrusion detection systems (or IDS from here on out) frequently uses deep packet inspection (DPI) to analyze traffic, which works by analyzing both packet headers and packet content. Historically IDSs are placed on the network perimeter, e.g. the border router. Many networks use a DMZ (demilitarized zone) in which public-facing services are placed, between the external router and the internal router that provides security to devices that are not supposed to be accessible to devices on the outside. In modern times, IDSs are placed within the network and not just on the perimeter, since users inside can also be responsible for (naively) installing malicious software. A relevant observation is that things like VPNs, bring-your-own-device, etc. help make the separation between the inside and outside of the network less clear.

Google's approach is to make every service accessible from the internet, but restrict access based on location/type of the device/user. This does away with the notion of even having an internal network to begin with. This approach of putting the security perimeter around individual services instead of the whole network is called the zero-trust model. It's too early to say if this solves all problems and it might be a little too large-scale to be of greater use to small organizations.

Of note is the difference between an IDS and an I*P*S (intrusion prevention system) is that the former is designed to detect and log intrusions, while the latter is designed to do something about them. An IPS might be able to block connections, for example.

There are different methods for detecting intrusions. Signature-based IDSs perform DPI by looking for byte-level patterns in payloads. This was initially based on fixed string matching (e.g. looking for long strings of "N"'s, since this is how the Code Red worm overflowed buffers), but was later expanded to use regular expressions. Regular expressions are now commonly used for DPI due to their ease of use and power. In reality things are more complicated, with considerations based on port, constant strings, etc, but it's a good enough simplified model. The disadvantages of signature-based IDSs are that they're limited since they have to process each connection separately, fragile since it's easy to make a signature no longer match, and possibly ineffective since many modern attacks cannot be represented well using regular expressions. For instance, attacks that don't use malformed traffic (e.g. HTTP session hijacking by sniffing authentication tokens) can't be detected using signature matching.

3.6.1 Parallelization

DPI takes time; the processor time available to analyze a DNS packet at 20 GB/s is about 200 cycles, but with Bro it takes around 4000 — about 20x longer. The conclusion is that parallelization is needed, with traffic split among many IDSs. If this is done naively, attacks that span multiple packets or connections may go undetetected. If IDS notes are allowed to share state there's a performance hit, so it may be necessary to split traffic in an attack-aware fashion.

3.7 Data recording

Once an attack is recorded, something has to be done with that information and determining what (if any) damage was done — forensics. Networking monitoring systems therefore need to be able to record data, but they can't do that with all traffic for privacy and storage reasons, so only relevant network data should be recorded. Interestingly, connection size is heavy-tailed, with only 12% of connections generating 96% of data.

One idea, then, is to record only a small amount of data (e.g. 20 KB) for each connection; small connections are recorded in full, and there's a reasonable characterization of large traffic. Other solutions like cloud storage of network telemetry data also exist.

3.8 Encryption

Data is frequently encrypted now, e.g. HTTPS, making DPI impossible without decryption. Ways around this always involve decryption. For example, TLS proxying relies on installing root CA certificates on users' machines; when a TLS session is established, the client sends traffic to the proxy which can decrypt it, then re-encrypts it with its own keys and makes the connection on behalf of the use. This is a form of man-in-the-middle attack, albeit with good intentions. It has the problem involving an all-powerful root CA certificate (which if compromised could end in disaster)a, and TLS interception boxes are frequently more out-of-date than the TLS stack of end hosts.

4 Transport

4.1 TCP

TCP (transmission control protocol) is used for reliable, stream-based transportation. Most things can't be sent on top of pure IP since there's no reodering, retransmissions might be needed, etc. Ultimately applications care about sending data and not packets, so a stream-based protocol makes sense.

TCP provides a communications abstraction that is reliable, ordered, point-to-point, and forms a byte stream. The protocol is implemented entirely at the ends, assuming unreliable, non-sequenced delivery. The basic reliability mechanism involves sending acknowledgments for numbered packets — if a sent packet is not acknowledged, the sender resends it until either an acknowledgment is received or the connection times out.

TCP packets contain source/destination port fields, a sequence number, an acknowledgment number, and a series of other flags and fields. There's a header length, advertised window, checksum, urgent pointer, and variable options in the remainder of the header.

4.1.1 Connection establishment

Hosts A and B must agree on initial sequence number selection, using a three way handshake. A sends a SYN packet to B with its starting sequence number, B responds with SYN/ACK and its sequence number, and A replies again with ACK. Using three packets is important

because each host needs to transmit its chosen sequence number, meaning a minimum of two packets to exchange the data, and three if you want to acknowledge the second one.

Sequence numbers are usually randomized. This helps to prevent confusion without respecting the quiet time (2 minutes), and it also helps prevent TCP hijacking.

4.1.2 Reliability

In the event a packet has been lost, it must be retransmitted. Hosts maintain an estimate of round trip time, or RTT, and wait at least one RTT before retransmitting. Correctly estimating RTT is important — if the RTT is too low, there will be unneeded retransmissions, but if RTT is too high, throughput will be poor. RTT estimation must also adapt over time, specifically using an exponential average system:

$$RTT_n = \alpha(RTT_{n-1}) + (1 - \alpha)(s)$$

... where RTT is the RTT at any given time, α is a constant with a recommended value of 0.8-0.9, and s is a new sample. The retransmission timer is set to $\beta \times RTT$ where $\beta = 2$. Every time the timer expires, the RTO is exponentially backed off.

4.1.3 Header Prediction

Header prediction is an optimization to receive processing on fast links. ON fast links, timestamps may be used to deal with segment wraparound. For every new packet, the receiver must check if the packet is in the window and if the timestamp of the packet is more recent than the preceding packet in the same windows. Header prediction is where, instead of checking if the new packet is in the window, check first if it is the next expected segment. This optimization is less important in modern times, especially since network adapters can perform LRO.

4.1.4 TCP Tahoe — Fast recovery refresh

In an early version of TCP called TCP Tahoe, packet loss causes cwnd to be set to 1 and slow start is initiated. In TCP Reno, losses detected by duplicate ACKs do not induce slow start, instead causing the protocol to enter a fast recovery state that does not involve slow start. In fast recovery, multiple packet drops within the window can cause the window to deflate prematurely. In new-Reno, outstanding packets are remembered at the start of fast recovery. If a new ACK is only a partial ACK, it's assumed that the following segment was lost and resent, so fats recovery isn't exited. This protocol was the default in Windows all the way up to Windows XP.

4.1.5 TCP Vegas

TCP Vegas included the first attempt to go beyond using loss as a sign of congestion. The idea is that when a connection is approaching congestion, pumping more data does not cause an equivalent increase in throughput. That is, the expected rate is $WindowSize/RTT$ but the actual rate is $Bytesinsegment/segmentRTT$. The congestion window linearly increased

when $Expected - Actual > L_B$ and linearly decreased when $Expected - Actual < U_B$. Other things changed include a more transmission policy using more precise timeouts, and a modified slow start.

4.1.6 TPC BIC & BIC/2

The motivation for this was that the TCP congestion window additive increase is too slow for links with high bandwidth and RTT. For example, fully utilizing a 10 Gbps path with a 100ms RTT takes a full hour. This is unattainable in practice because no network can guarantee an hour of hour free transmissions at 10 Gbps. The proposed solution is to adjust window growth rate based on network conditions.

In BIC/2 the core idea is to combine binary search increase with additive increase. A binary search between the window size after a loss and the window size after fast recovery is performed. If binary search initially requires a sudden increase above a threshold S_{max}, additive increase is used (by S_{max}). This was the default in Linux until 2006.

5 Tunnels, NFV, and Middleboxes