CSCI 3171

Protocols and More

McAllister

Alex Safatli

Monday, February 18, 2013

Contents

More on the Application Layer

Peer-to-Peer Applications

Sample applications of the peer-to-peer sort include:

- distributed hash tables (DHT) — indexing structure,
- file transfer (BitTorrent) — distribution of work among peers,
- Skype — supernodes.

These last two will be looked at only at a cursory level (Skype is not very well-documented, for instance).

Distributed hash tables were created to solve the problem of seeing what peers have what distributed where; it is an *indexing structure*. Given a set of peers and resources to store at peers, how can we locate which peer holds what resource?

What happens is we assign an integer identifier to each peer in range $[0, 2^n - 1]$. Each identifier can be represented by n bits. This requires each resource key to be an integer in the same range (hash distinctive information about the resource to get a key). For example:

```
key = hash("Led Zeppelin IV");
```

Note the issues that arise:

- we have to find host x,
- we have to store the table somewhere,
- and hosts could disappear.

The central issue is assigning `(key,value)` pairs to peers. The rule is typically to assign keys to the peer that has the closest ID. The convention we will use in lecture is that the closest is the immediate predecessor of the key. For example, if keys are 4 bits and peers 1, 3, 4, 5, 8, 10, 12, and 14 are represented. If the key is 13, then the predecessor peer is 12. If it is 0, the predecessor peer is 14.

This is a model of **circular DHT**, the most common type of DHT: each peer is *only* aware of immediate successors and predecessors. Therefore, what happens is the table is not being hosted anywhere but any given host is storing a single record. This leads to the necessity of a **recursive search** with $O(N)$ messages on average to resolve a query when there are N peers — if you hit yourself, you know not to answer your own query and you know the hash is invalid. Remember that unicast messages to previous contact, etc. can still be utilized here.

Therefore, we could make this more efficient: keep track of *shortcuts* — perform a binary-like search such that each peer keeps track of IP addresses of predecessor, successor, and shortcuts; this results in $O(logN)$ neighbours and $O(logN)$ messages in query.

> **NOTE** Assuming a full set of hosts of, say 15, a peer 1 would keep track of IP addresses for 0, 2, 3, 5, and 9. Similarly, 9 would keep track of addresses to 10, 11, 13, and 1. Each node will have a logarithmic number of pointers. Having a constant-N time (with knowing all pointers) defeats the purpose.

When a peer enters, it has to notify respective peers it now exists and populate its list of IP addresses. To handle **peer churn** (a peer leaving), each peer is required to know the IP address of its two successors. Each peer can therefore periodically ping its two successors to see if they are still alive ("*keep-alive/heartbeat*" message).

Note that any given peer will have a backup copy of its data at its predecessor. The amount of backing up that is done is dependent on the data being stored.

Peer-to-Peer Applications continued on next page...

THIS SYSTEM IS ONLY MEANT TO BE AN INDEX. All files are not stored; IP addresses of the machine that has the file are stored — it is only meant to help find the file itself. Not a lot of data will need to be stored. Files are not being physically backed up to peers.

File distribution has an aim to reduce transfer time of a file of size F to N users where user i has to upload bandwidth u_i and download bandwidth d_i. Server has upload bandwidth u_s. See slide to see how long it will take to distribute files. Multicast creates smaller groups of addresses – it is difficult to maintain; hosts have to be aware and register on its own.

BitTorrent is a process of sharing 256 kilobytes of chunks of a file between peers. The receiver coordinates reassembly on the chunks into one file. It is a *hybrid architecture*: a **tracker** keeps tabs on the peers that have a piece of the file (a peer first contacts a tracker to contact peers with the file), peers exchange chunks directly, peers may come and go, and once a peer has a full copy, he can leave or become a "seed" to continue distribution.

SHARING CHUNKS. On first contact, the initiating peer may have no part of the file. As a peer receives chunks of the file from others, it offers those chunks to other peers *in parallel* with receiving further chunks. Peers periodically share with neighbours their list of chunks. Chunks are often distributed on a rarest-chunk-first basis to increase their availability.

NOTE In peer-to-peer systems, we can offer incentives to keep sharing. In a client-server model, the server pays a massive amount of resources to stay running — how is the server given an incentive to incur this cost? Could be partly advertisement, fee, to illicit a reaction, to share information.

We can also talk about the concept of **tit-for-tat sharing**. Each peer has a set of preferred peers for the download. Peers periodically evaluate how much each other preferred peer is helping with their download. Peers that help a lot receive more upload bandwidth from a given peer. Finally, peers periodically choose some non-preferred peer to test as a "preferred peer" — giving the peer more upload bandwidth and see if that peer responds. This incentivizes people who share.

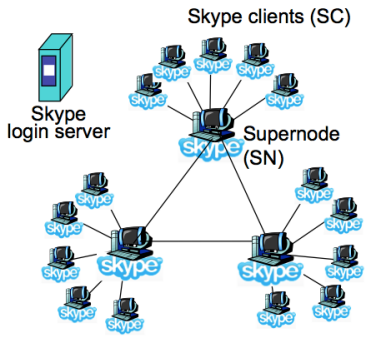

Figure 1: *A hypothetical Skype supernode hierarchy.*

Skype is a case study in peer-to-peer networking. It is *inherently* peer-to-peer where pairs of users communicate. Note that it is a proprietary application layer protocol (this is inferred *via* reverse engineering). It involves a hierarchical overlay with **supernodes** (SN) and an index maps usernames to IP addresses; these are distributed over SNs. Therefore, it is not *entirely* peer-to-peer – clients that are reliable and have good bandwidth, etc. are identified as supernodes, acting as a proxy for transient clients or hosting the hash table or index structure.

The Transport Layer

Overview

Transport layer topics include:

- multiplexing and demultiplexing,
- error-free delivery (integrity),
- reliable data transfer (in-order delivery & guaranteed delivery),
- flow control (keep sender from overwhelming receiver),
- and congestion control (keeping amount of traffic appropriate to what network can handle).

This will not deal with or include with *bandwidth guarantees* (depends on the past; transport layer does not care about the past) and *delay guarantees* (does not control delay). Overall, the transport layer PROVIDES A LOGICAL CONNECTION BETWEEN TWO APPLICATION PROCESSES; the processes are distinguished by the combination of their IP addresses and port numbers. Processes on the same host are distinguished by their port numbers, and on different hosts they are distinguished first by their IP addresses.

The transport layer offers two levels of service.

1. **UDP**: has *error detection* only,
2. **TCP**: has *error detection*, *reliable delivery*, *flow control*, and *congestion control*.

There is no notion with a connection in UDP (**connectionless**) whereas TCP has the notion of some manner of a virtual circuit in terms of streams of bytes travelling from one process to another (**connection oriented**).

In connectionless multiplexing and demultiplexing, we uniquely identify a connection by destination IP and port. In connection oriented multiplexing and demultiplexing, both sides become important: source information is necessary. Two sockets are necessary: one for each communicating source.

Basics of Error Detection

The sender and receiver agree on a function $f(x)$ where if the sender wants to send a message M, they send M and $f(M)$. The receiver would receive message M' and value v — they compute $f(M')$ and if $f(M') == v$, the receiver assumes there were no errors with the transmission. This process allows for false positives, but *not* false negatives. This value v is called a **checksum** in UDP.

Therefore, this is typically a trade-off of: size of the output from $f(x)$, complexity in computing it, and the probability of detecting *types* of errors.

Other factors to consider are: the number of errors that are detectable (think *parity bits*), the ability to invert $f(x)$, and specific error patterns in the medium (e.g. bursts of bit errors, specific bits prone to error, as an error with an address bus).

Types of error detection functions include: repetition, parity (used for memory systems), sum of digits (checksum; UDP, TCP, IP), polynomial operation (cyclic redundancy check CRC – ethernet), one-way hash functions (digest), and data expansion to add redundancy or separation.

The Transport Layer continued on next page...

User Datagram Protocol (UDP)

This is the MINIMAL transport layer protocol. It offers only **error detection** and **best-effort delivery**. It does not handle:

- flow control,
- congestion control,
- in-order delivery, or
- guaranteed delivery.

This is more week than is necessary; only the minimal amount of work is put in by this protocol. The only reason error detection came in was out of concern for the protocol itself; it has a header, and to do its minimal-effort job properly, that header should be okay.

It is a **connectionless** protocol and so all segments must be self-contained: source/destination ports, data and length indicator for data, and error detection information. All UDP segments are handled independently of all other UDP segments.

WHY UDP? It has:

1. *minimal space overhead for the header*,
2. *no connection set-up overhead*,
3. *no sender and receiver state* to maintain,
4. *no delay in processes received packets* if they are received out-of-order, and
5. *no restriction on using bandwidth* (good for some applications where you want to send a lot of data at once in a small window, but not so good for the network so use wisely).

For a little bit of reliability, UDP can be employed and reliability can be ensured on the server- or client-side (e.g. they take part of the responsibility at the application layer). The header for UDP only has 8 bytes of information. It is often used for streaming multimedia applications (loss tolerant, rate sensitive). A lot of protocols, like UDP, will take about **octets** – the protocol structure is made up of 8-bit chunks of size 4 in length.

Note that the **checksum** is only 16 bits long. For a large amount of data, this may be problematic. We want to map a large amount of data to a smaller space. This relates to the fact that there may be false positives.

Reliable Data Transport

NOTE See paper notes for more checksum information.

RDT 1.0, **RDT 2.0**, **RDT 2.1**, **RDT 2.2** are all techniques and general protocols of **reliable data transport**. This is a general overview of how things may be done in order to ensure reliable data is transported. The concept of acknowledgement has been added in order to introduce methods of error correction. A finite state machine was developed for each of the sending and receiving aspects of of reliable data transfer.

HOW CAN WE HANDLE LOST PACKETS? Retransmitting the packet if we know that it is lost. HOW DO YOU DETECT A LOST PACKET? No response after a "reasonable" amount of time. The time to wait requires tuning (depends on sending and return time), and the issue is if data was delivered and the ACK was lost, we send the data twice.

Reliable Data Transport continued on next page...

Furthermore, if the ACK is just delayed, we send the data twice and may receive two ACKs back. RDT 2.2 already filters out duplicate data.

This introduces **RDT 3.0** and only involves a small number of modifications. A notion of detecting or measuring time is added when data is sent. When an acknowledgement is received, the timer is stopped. *what do we do if we are waiting for an acknowledgement and receive the wrong acknowledgement*? This is debatable. The timer could be stopped and restarted (because new data is being sent out), but what we normally do is let the timer expire. If a timer expires, the packet is sent again and a new timer is started.

This revision is what we call a **stop-and-wait** protocol. Performance is gated by the notion of receiving an acknowledgement within a certain range of time. Expiry of a timer forces another message to be sent and a new timer to be started. An issue with stop-and-wait is waiting for the ACK before sending more data leaves the channel *idle*.

The solution is to allow for pipelining of data (send more than one packet at a time). This introduces new problems:

1. How do we track which data has been acknowledged (selective ACK, go back N)?
2. How do you keep from just flooding the network with as many packets as you want (flow control)?

Pipelining allows up to n packets to be unacknowledged at any one time. The efficiency for this is: $n\times$ transmit time /(RTT + transmit time). This is a factor of n increase (up to filling all the idle time) — need more sequence numbers than 0 and 1. The sender must remember all n packets until they are acknowledged — consumes resources.

RESOURCE CONTROL AND PIPELINING. The sender tracks ranges of contiguous sequence numbers that it is willing to have left unacknowledged. The *size of the window* dictates the *maximum commitment* from the sender on storing unacknowledged packets.

When an acknowledgement arrives at the sender, he can shift the window of all *valid sequence numbers*. This is called the **send window**. Once all of the sequence numbers in the send window are used, the application layer cannot send more data until some ACK is received, but this may not necessarily force a move unless it can without losing track of something. Therefore, this is a trade-off between best use of the channel and easy programming. This window tends to lag behind the receive window.

How do we deal with acknowledgements in pipelining? Recall above.

- **Selective ACK**: Every packet is acknowledged *individually*. The receiver must store all packets that it receives, regardless of order of receipt. Used when sending data is expensive.
  - The receiver also then tends to maintain a list of sequence numbers for which they are willing to receive or buffer packages. *Data has to be stored and maintained here.*
  - This is called the **receive window**; if |send window| $\leq$ |receive window|, the receiver cannot be overrun, but we may want this for our protocol anyway (flow control). We need as much as twice as many sequence numbers as there in the window. Note that any ACKs outside the window are ignored.
  - A smart implementation will move an item up as soon as it has been received and the window has moved (e.g. there is a message in the buffer).
  - One must TRACK the following: the start of the window, the size of the window, the next available sequence number, and for each sequence number in a window, whether or not it has been ACK'd.
  - When an ACK is received: mark just that sequence number as ACK'd, shift the send window so that the start matches the first unACK'd sequence number.
- **Go back N**: Acknowledge packet z implicitly acknowledges previous to and including z (called a **cumulative ACK**). The receiver can choose to remember just the packets received in order at the expense of the sender retransmitted the out-of-order packages.

- One must track: the starting point of the send window, the size of the window, and the next available sequence number in the window.
- When an ACK is received, move the start of the window to the packet after the ACK'd number. Again, any ACK outside the window is ignored.

Note the use of **timers** for this. If ONE TIMER WAS PRESENT FOR EVERY PACKET, there are **many timers to track**. The operating system's implementation will often multiplex multiple timers over a single timer. When a timer expires, re-send only the packet whose timer expired. Therefore, the receiver *must* store all the packets it receives (normally, it is not obliged to do so), regardless of the order in which they are received. This is used most often for selective ACK.

NOTE By multiplexing, this means the single OS timer is set to the lowest timer and gives the illusion there is more than one timer.

On the other hand, **a single timer could be present for all packets** (usually the oldest packet). This means there is only one timer to track. When the timer expires, re-send all unACK'd packets. The receiver *need not* store out-of-order packets. When it receives an out-of-order packet, it sends an ACK for the last item sent in the correct order (knowing the sender will re-send all subsequent packets). When the oldest packet is ACK'd, restart the timer for the next oldest. This is used most often for Go back N.

SELECTIVE ACK *v.* GO-BACK-N. **Selective ACK** conceptually uses many timers, has greater storage requirement for the receiver, minimizes the number of duplicates sent, and is best used when packet loss is uncorrelated between packets. **Go back N** has less storage requirements on the receiver (a receiver can choose to store out-of-order packets but it is not necessary). A delay in a packet near the start of the send window re-sends many duplicate packets. It is best used when losing one packet means that other packets are also likely lost.

See table on the last slides for RDT.

Transmission Control Protocol (TCP)

This is a **point-to-point** and stateful connection protocol. It is only between two parties and is known as a **connection-oriented transport layer protocol**. It makes sense to broadcast UDP, but not TCP. It models a **stream of bytes**; no message boundaries or direct concept of a "message". It features:

- error detection,
- reliable data transmission (guaranteed, in-order delivery),
- flow control (making sure sender and receiver are happy),
- congestion control (making sure network is happy), and
- uses a "Go back N"-like cumulative ACK.

It also has **duplex data transmission**: bi-directional data flow. Each side has its own **send window** and **receive window** (corresponding).

TCP coordinates state; explicitly, starting sequence numbers and receive window sizes are coordinated. Implicitly, or a **sensed state**, manages the network (congestion in the network) — there is no way for TCP to know directly what is happening in the network later, but it can guess with a few clues such as the time it takes to send a message or a loss of packets.

NOTE In socket code, opening a socket, closing it, and trying to open it again will sometimes fail. Have to wait for the TTL of all messages is enough for all messages to expire in the network — decreases the probability of a collision.

A **handshake** is also used to initiate a connection (three-way; need three messages to go back and forth). Finally, we also have a two-way tear-down of connection: ensures that both parties are done sending data on the bi-directional channel. Every conversation is started with a different packet sequence number to ensure old conversation messages are not received.

An **initiator** and **responder** is present when setting up a connection. There is first a **SYNchronize** of the peers (often called a SYN request). Each side proposes a starting sequence number for the data it will send (which triggers the other to set up their receive window to correspond with this send window) and acknowledges both the SYN request and the sequence number of the *next byte* it expects to *receive* from the other.

See slide on *TCP segment structure*. Sequence numbers count by *bytes of data in the stream*, not segments or messages. `head len` counts the number of 32-bit words in the header. `R, S,` and `F` are used as establishing and managing the connection (setup, teardown commands) and `A` identifies whether or not the ACK# is valid.

TCP implementations can buffer some stream data while waiting for more to arrive to make the best use of network resources. The protocol allows for the user to request an immediate sending of data (push) and the identification of that data part-way through the packet needs to be considered quickly (**urgent data**). The urgent data pointer points to where the urgent data is sitting in the rest of the message.

TCP tear down involves each peer of the initiator and receiver sending a **FINalize** message to the other and an **ACKnowledging** of the closing. This is often called a FIN message. One side does not need to close its side its side immediately if it still has data to send.

In the TCP state machine, the **established state** is regular operation. We will never have to replicate this picture! But, the various steps are not all that complicated in the end. An *active close* is when the current peer is trying to initiate a FIN, while *passive close* is when the other peer has closed the connection.

PIGGYBACKING ACKS. A segment with data can also **piggyback an ACK** for past data. Recall in RDT, whenever we send off data, we require an acknowledgement back. In TCP, we do not require ACKs to be their own messages; otherwise, a lot of data is being sent around that is meaningless. TCP can delay an ACK for a short time to wait for some data on which:

- to piggyback on, or
- to cover multiple packets with one ACK.

TCP AND RELIABLE DATA DELIVERY. Segments are typically **pipelined** (no longer gating or slowing down the sender on a one-segment-to-one-segment basis. It uses a cumulative ACK and a **single retransmission timer**; when a timer expires, we re-send the segment attached to the timer (the oldest segment) and not all unacknowledged segments. In reality, the timer is typically 1 and a half to 2+ return times. In reality, they are just over 1 round trip time (RTT); but there is a problem with this — the RTT varies and nobody can say what one RTT is. The solution to this is estimating the RTT on-the-fly based on the traffic being seen.

Estimating the RTT uses an **exponential weighted average** where EstimatedRTT $= (1-\alpha)\times$ EstimatedRTT $+\alpha\times$ SampledRTT where $0 \leq \alpha \leq 1$. This smoothes out anomalous delivers; with a large α, the most recent samples are trusted the most. With a small α, historical network timing is relied upon. α is typically taken to be $\frac{1}{8}$.

NOTE Note that this will typically mean if half of the transmissions are below the estimated RTT, they will be timed out. What is the best answer to this? We want a buffer zone; if we want to use a Gaussian distribution to track 99% of the values, then this is a possible solution. However, computing square roots are expensive operations (there is no instruction set for it) to calculate the standard error. What if we want to *estimate it*? That is shown below.

Furthermore, the variance in time is approximated by using formula DevRTT = $(1-\beta) \times$ DevRTT $\beta \times |$SampledRTT $-$ EstimatedRTT$|$. β is often taken to be $\frac{1}{4}$. Therefore, the **timeout** is set to EstimatedRTT + $4 \times$DevRTT. If the network gets faster, the deviation will shrink; this is an adaption to the situation.

NOTE On the initiator side, the size of the send window is the size of the **buffer** *from* the application layer. Its receive window is the size of the buffer *to* the application layer. The former is a choice of the operating system while the latter is a choice of the transport layer or operating system — together, these are tradeoffs of the implementation.

Sending data involves flow control — the bytes to send is the minimum of:

- **SendWindowSize** (nextSendSequNumber − sendBase),
- **ReceiveWindowSize** (from latest ACK packet; space available at receiver),
- **CongestionWindowSize**

Duplicate ACKs are dealt with by re-sending the oldest segment if 3 duplicate ACKs are received. This is typically a sign of traffic getting through but there is something the matter. This retransmission called a **fast retransmit** since we are retransmitting before the timer expires.

Otherwise, this is a **hard retransmit** (when timing out). Furthermore, fast retransmission would trigger the congestion window to probably be decreased in size (halving). Timing out would trigger a decrease in size to a minimum.

Each of sending data, timing out, or duplicate ACKs can change the size of the send window.

Congestion Control

Results from one or more network devices receiving too many messages to process results in **congestion**. This could be a single host overwhelming the network or a combination of many well-meaning hosts all sending little data but their combined volume is too large.

Congestion control operates on a unit of maximum segment size MSS — this can change depending on the local link layer technology. This section will focus on managing a congestion window in parallel with the send and receive windows.

HOW DO WE DETECT CONGESTIONS? The *network elements* know based on their queues and when they drop packets (**network-assisted congestion control**). Historically, no way to let the communicating nodes know. Explicit Congestion Notification (ECN) allowed in 2001 but still not widely deployed (a flag). Of course, this means that the network devices must be willing to report this. The *sender of packets* (on both ends; **end-to-end congestion control**) can recognize **long delays** (seen as duplicate ACKs) or **lost packets** (seen as timeouts).

Consider two hosts that share a router equally with a common data transmission rate T_x. The rate of receiving non-duplicate data at the receiving end is R_x. If the router has infinite memory capacity, then there is no loss. Otherwise, there will be loss. Re-sending data to guarantee delivery *further* congests the network. Multi-hop performance makes the performance curve even worse.

Congestion Control continued on next page...

NOTE The start of the congestion window is always the start of the send window, but the window thereafter is what is adjusted.

The goals to congestion control are to:

- **reach the peak** of the curve quickly (SLOW START),
- try to **stay balanced** around the peak once we reach it (AIMD),
  - if more bandwidth becomes available, we want to use it,
  - and if other TCP connections arrive, we want to back off to allow bandwidth for the new connection.
- give the network **time to recover** when congestion happens (be conservative after timeout events), and
- **consume available network** capacity, when available (sharing with others fairly).

AIMD is short for **additive increase, multiplicative decrease**. It is a continual probe for more bandwidth (increases the size of the congestion window by 1 MSS each RTT) but when backing off when too much bandwidth is used, the size of the congestion window is halved (after receiving 3 duplicate ACKs). This is often known as a **congestion avoidance** phase.

NOTE See graph that was drawn on board, and read e-mail correction.

Slow start will influence this, however. AIMD can be slow in reaching the initial top bandwidth if we start with a small congestion window. On the other hand, we can overwhelm the network if we start with a large congestion window. Slow start begins with a small congestion window and then grows it more aggressively than AIMD:

- **initial congestion window** is 1 MSS,
- **double the size** of the congestion window every RTT, and
- when **packet loss** happens, revert to AIMD.

A **timeout** generally indicates *severe congestion* in the network. The congestion window size is adjusted to give the network time to recover. Therefore, $\frac{1}{2}$ of the congestion window size is set and remembered as a **threshold**. Then, all congestion windows are reduced to 1 MSS where slow start is operated until the window size matches this remembered threshold — once there, AIMD is reverted to.

NOTE Fast retransmit occurs when packet loss begins to occur. Messages are resent immediately — see above.

Nothing in the network prevents not adhering to this TCP protocol. However, not working properly will interfere with proper communication — what stops one from doing this is the client themselves and how well others play with them.

NOTE UDP is *not* TCP-friendly.

If we assume that the maximum transmission rate remains fixed at W, the RTT time remains fixed, and TCP remains in *congestion avoidance* mode (an oscillation is occuring) then the **throughput** is (when two users are present):

- lowest at 0.5 W/RTT,
- highest at 1.0 W/RTT,

- and linear increase therefore means average throughput is around 0.75 W/RTT.

NOTE We are describing *TCP Reno*; whenever a timeout occurs, we go back to the start (no duplicate ACKs). When duplicate ACKs occur, fast retransmit happens.

TCP fairness refers to the property of TCP to share bandwidth *equally* between all connections using the same link. Given a shared link with bandwidth R and with k users, a protocol is fair if each user gets a bandwidth of approximately $\frac{R}{k}$. This is a **convergence point** for AIMD operation.

Considering two hosts transmitting using AIMD and sharing one link, we will assume that one hosts starts with more bandwidth than the other. This happens if one connection starts *later* than the other. See the slide and graph on that slide for the effect of AIMD converging upon an equal use of bandwidth. How fast does this convergence happen? As decreases occur, after $log_2(a-b)$ steps, the difference series $a-b$, $\frac{a-b}{2}$, $\frac{a-b}{4}$, ... will converge to 0 (same window size will come about).

Note that fairness depends on the congestion control mechanism. UDP has no congestion control and therefore a UDP connection can eventually *starve* TCP connections. **Fairness** applies on a *per-connection* basis. *Nothing prevents an application from opening parallel TCP connections to get a bigger piece of the bandwidth.* What prevents someone from implementing TCP without congestion control? With slow start at all times? Without the (immediate) multiplicative decrease?

See the slide on transport layer topics. A good question to ask yourself is whether or not it is possible to have an intermediate protocol that only uses half of the feature resources from TCP.

Network Security

Security Goals

This section of the notes will cover the following questions.

- what do we consider to be **network security**?
- what are the basic tools of **securing information**?
- the application of securing information, and
- **Secure Socket Layer** (SSL; encryption right above the transport layer) and **Transport Layer Security** (TLS).

An interesting acronym of security is CIA: **confidentiality**, **message integrity**, and **availability**. Confidentiality is the concept that only the sender and intended recipients can understand message contents. Message integrity involves how the recipient receives the exact message the sender sent. Finally, availability refers to services that are available to users.

Other goals of security involve **authenticity of the sender and receiver** and **non-repudiation**. Authenticity refers to the idea that the sender and receiver can confirm each others' identities. Non-repudiation is the idea that once a party issues a message and it is received by another, the sending party cannot later claim it never sent the message.

Network Attacks

Basic **network attacks** involve actors **Alice** and **Bob**; these are two people who want to communicate. Then, there will be someone who wants to do bad: **Oscar** (opponent), **Eve** (eavesdropper), or **Trudy** (intruder). Their environ-

Network Attacks continued on next page...

ment will be the place where Alice and Bob are communicating over an unencrypted channel to which Trudy has access.

What can Trudy do?

- **Eavesdrop**: intercept messages to learn their contents,
- **Insert messages**: add messages to the conversation,
- **Impersonate**: pretend to be one or the other (Alice or Bob); fake the content of any field,
- **Hijack a connection**: take over an existing connection by substituting her information for that of Alice or Bob (extreme version is **man-in-the-middle** attack where Trudy acts as a relay between them),
- **Deny access to a service**: prevent a service from being used; could just remove packets (but the network will route around it as a failure), force a server to commit resources unnecessarily, or overwhelm a server with too many fake requests.

Tools that can be used to increase security include:

- security through obscurity (not a viable option),
- shared secret information (must prevent brute-force attacks at the least),
- computationally difficult problems (base work on NP-hard problems so it is computationally impractical — discrete log in Duffie-Helman or factoring), and
- information theory (base work on which information is delivered to anyone else; how the messages line up with each other, etc.).

Early Ciphers and Symmetric Key Cryptography

Cryptography often uses keys (where an algorithm is known to everyone; only the keys are secret). This includes **hash functions**, which involve no keys and has nothing secret but is often used to provide message *integrity* rather than hide information, **symmetric key cryptography**, which involves one key and is often called private key cryptography, and **asymmetric key cryptography**, which involves two keys and is often called public key cryptography.

Early ciphers include the **substitution cipher** (e.g., Caesar's cipher or ROT13) where the key space size is 26!, **polyalphabetic ciphers** where n monoalphabetic substitution ciphers are used (e.g., Vigenere cipher and Autokey cipher), and **one time pad** where the sender and receiver have an agreed-upon sequence of random bits or letters longer than the message length.

NOTE Exchange message from Assignment 3 is analogous to SSL where there are different encryption functions built-in. RC4 is an option for SSL.

Symmetric key cryptography involves an encryption and decryption key that are the same — to share the initial secret, a key has to be exchanged in person, use a different encryption system to exchange the key, mutually compute a key together, or get a key from a trusted third party. These are usually *far* faster than asymmetric key cryptographic systems. These include **stream ciphers** and **block ciphers**. A popular stream cipher is RC4 — it is considered good, a key can be from 1 to 256 bytes, and is used in WEP for 802.11 and can be used in SSL.

Block ciphers processes the message in blocks of k bits. A 1-to-1 mapping is used to map k-bit block of plaintext to k-bit blocks of ciphertext. How many possible mappings are there for $k = 3$? In general, this is $2^k!$ mappings; this is huge for $k = 64$. Using a table is not ideal here, so a function is used to simulate a randomly permuted table.

A **Fiestel Cipher** is a general format for a block cipher algorithm; it repeats a set of operations in rounds where the data is split into pieces, a prototype is applied to one of the pieces, other parts of the data is mixed in, and the order of the data is swapped for the next round. *Why rounds in prototypes*? If there was only a single round, then one bit of input affects at most 8 bits of output. In the 2nd round, the 8 affected bits gets scattered and input into multiple substitution boxes. This becomes less efficient as n increases.

Why not just break a message in 64-bit blocks, encrypting each block separately? If the same block of plaintext appears twice, it will give the same ciphertext — this is called an **Electronic Code Book** (ECB). This leads to one form of an attack and essentially puts the entire system into a table. Depending on how much structure there is, this is a viable option. A greater amount of security involves generating 64-bit random numbers for each plaintext block and mixing the data. This is inefficient and requires the sending of the random number and ciphertext.

Cipher Block Chaining (CBC) generates its own random numbers and has encryption of current block depend on the result of the previous block. The first block is encrypted with an **initialization vector** (IV) which does not have to be secret. The IV is changed for each session or message and guarantees that even if the same message is sent repeatedly, the ciphertext will be completely different each time. This cannot be parallelized and is therefore serial – it is slow.

Data Encryption Standard (DES) is a symmetric key Fiestel cipher with 16 rounds of encryption. It features a 56-bit symmetric key (32-bit subsets are used as keys during the Fiestel process) and 64-bit plaintext input. It is a block cipher and utilizes CBC. How secure is DES? It did not have a known good analytic attack and required about a days' worth of brute force computation to break the 56-bit key encryption phase. It is no longer the standard. 3DES (encrypting 3 times with 3 different keys: encrypt/decrypt/encrypt) to make DES more secure.

The successor to DES is the **Advanced Encryption Standard** (AES). It is a new (November 2001) symmetric-key NIST standard, processes data in 128-bit blocks (with 128, 192, or 256-bit keys) and brute force decryption taking 1s on DES is estimated to take 149 trillion years for AES.

Asymmetric Key Encryption

Asymmetric key encryption have *encryption* and *decryption* carried out using *different* keys. The encryption key can be publicized and distributed widely for anyone to use, but the decryption key is kept secret so only the intended recipient can decrypt the data. Consequently, COMMUNICATION IS ONE-WAY: the sender of a message cannot decrypt a message they just encrypted. This only came about in the mid-70s.

> **NOTE** Taking some text and breaking the code or decrypting it will not be done on any tests. The basic idea behind them and how they flow is the objective of this course.

The encryption key for A is typically denoted as K_A^+ and the decryption key is typically denoted as K_A^-. Knowing the public key K_A^+ should not allow anyone an advantage in computing the private key K_A^-. The message can be computed using $K_A^- \times (K_A^+ \times m) = m$. The inverse *could* also happen: $K_A^+ \times (K_A^- \times m) = m$.

Examples of asymmetric key encryption include:

- **RSA** (relies on difficulty of prime factoring; operates over modulo arithmetic),
- **El-Gamal** (relies on difficulty of discrete logarithms; operates over a cyclic group with inverses — integrates modulo a prime; elliptic curve group), and
- **Chor-Rivest Knapsack** (relies on the difficulty of the subset sum problem; knapsack problem — have a whole pile of objects and a knapsack with finite volume... want to fill the knapsack up to try and maximize the value, etc.).

Asymmetric Key Encryption continued on next page...

RSA interprets the message as an *integer*. The sender and receiver agree on a maximum message size n; ECB or CBC is used for longer messages (see above). The encryption and decryption keys are specially-chosen integers e and d. To encrypt a message m: $m^e \bmod n = c$. Decryption is: $c^d \bmod n$. The order of use of the encryption and decryption keys does not matter.

Breaking RSA is equivalent to factoring n: an NP-hard problem. The underlying math is picking two big primes p and q where $n = pq$ and pick e and d such that $pe + dq = 1 \bmod (p-1)(q-1)$. Note that NP-hard is relative to a reference model computer (Turing machine): but how about *quantum computing*?

Elliptic curve cryptography involves elliptic curves which are represented by the points at integer coordinates on an equation of the form $y^2 = x^3 + ax + b$ where there are specific rules set for computing the sum of two points: $M_3 = M_1 + M_2$.

> **NOTE** A document from the NSA (2009) shows that for an elliptic curve, a 160-bit key provides as much security as a 1024-bit RSA or Diffie-Hellman encryption key. 521-bit corresponds to a 15360-bit key in RSA.

El-Gamal encryption has preparation which selects a cyclic group G of order n with generator α. A random integer a, $1 \leq a \leq n-1$ is chosen and α^a is computed. The public key is therefore (α, α^a) and group G (which could be an elliptic curve group). Encryption represents the message as an element m of G where a random integer k is selected, $1 \leq k \leq n-1$. $\gamma = \alpha^k$ and $\delta = m \times (\alpha^a)^k$ are computed. The ciphertext is (γ, δ). Decryption uses a private key to compute γ^a and then γ^{-a}. m is recovered by computing $(\gamma^{-a}) \times \delta$.

Realize that asymmetric encryption can be at least 100 times *slower* than symmetric encryption. RSA requires exponentiation of arbitrary-sized integers while DES and AES only require bit-level operations. Therefore, asymmetric encryption is often used to **establish a secret key** for a short period of time between two parties — a **session key** that is used to encrypt all data in the conversation.

Realize that given a cipher, not all keys are necessarily good. A **weak key**, used with a specific cipher, makes the cipher behave in a way that is not desirable. Once these weak keys are characterized, checks are usually done when keys are generated for these weak keys.

Key Distribution

Symmetric keys are typically distributed as pre-shared keys, through a trusted third-party (**key distribution centre**), through Diffie-Hellman key exchange, or an exchange under asymmetric key encryption.

Asymmetric keys have their public keys publicized as wide as possible (web site, email, open access server, ...) and provide digital signatures on publically-distributed information (**certificate authorities**, **web of trust**).

A **key distribution centre** (KDC) has a symmetric key with A and a different symmetric key with B: keys K_A and K_B. Therefore, A asks the KDC for a session key with B and the KDC generates a random session key K_s; this is sent to A and B under separate encryptions and A and B now communicate using this session key K_s. This entirely allowed two people to have a single symmetric key.

Certificate authority (CA) ensures public keys are trusted — this is for asymmetric key encryption; the CA encrypts B's public key K_B^+ with a decryption key K_{CA}^- to make $K_{CA}^-(K_B^+)$ public. This is not like a KDC where the KDC always has to be there. Later, K_{CA}^+, their public key which could be hard-coded or vouched for by another CA, is used to get the original key back. See slide. This ensures that the CA is the one sending the key.

Web of trust is a model for verifying encryption keys of others. Individuals can digitally sign one another's encryption keys (often meeting in person to verify identity). When someone els's encryption key is retrieved, the signatures of others are presented (who have signed that key). You look through who signed the key to find anyone who you already trust; if those are valid, your trust in the key increases. The more people that you know who sign

Key Distribution continued on next page...

the key, the more you trust the key. You can trust some people more than others (their signatures weigh more in the verification). The idea is that this graph should be connected (completely distributed).

Message Integrity

Message integrity is now something we can discuss: we have a foundation of all tools we can use to ensure this. Message integrity involves a number of components:

- **message contents have not been altered** (error detection for accidental changes, **Message Authentication Codes** — MAC for deliberate changes),
- you can **confirm the source of the message** (source identification via MAC or certificates),
- it has **not been replayed** — think: man in the middle (**nonces**; numeric once — should only be acted on once and not any time later), and
- it appears in the **right sequence** (sequence numbers).

Message digests are found in error detection with a more complex detection function: a hash function. The hash value for a message is called its message digest. A hash function is a many-to-one function, so several messages could have the same message digest. Properties of the hash function include:

- easy to calculate,
- **one-way** — not invertible such that you can reconstruct the message from the digest,
- **collision resistant** (not invertible) so you cannot reconstruct other messages that have a specific digest, and
- output is random — spreads digests across a digest space and uses all digests.

> **NOTE** The **birthday paradox** involves a heightened probability of two people having the same birthday in a room with n people. This matches up to hash functions. To find two documents with the same hash value is analogous to this problem — induction of a collision in the hash table. Therefore, have to make sure that collisions are not too similar.

Common hash function algorithms include **MD5** (128-bit message digest), **SHA-1** (USA NIST standard, 160-bit message digest), and **RIPEMD** (160-bit message digest; learned from SHA-1).

Once these digests have been obtained, we want to verify the contents of the message have not changed and verify the sender. Therefore, a one-way hash of the content and prepended shared secret to the message is computed — only people who know the secret will get the right hash value and people who know the secret can verify the sender also knew the secret. This is the notion of **message authentication codes** (MACs) (this computation).

A **hashed MAC** (HMAC) is an updated version of a MAC to add a fair bit more security. It does multiple hash computations: prepends secrets, hashes the result, prepends secret to digest, and hash that result to get a HMAC. Hashing a hash value stops **extension attacks**.

The MAC ensures that the person who created the message knows the secret; it does not say that the person who created the message is the person sending it (this is therefore open to replay attacks) and does not say which person created the message if more than one person knows the secret (open to repudiation). If asymmetric key encryption is used to provide **authentication** (each person keeps their private key private), this ensures they are the only person who could have computed a key.

Simple authentication involves encrypting the whole message with a private key; anyone can decrypt it using a public key and provides no confidentiality to the message. A meaningful message only comes out if the person

Message Integrity continued on next page...

who holds the matching private key did the encryption.

Rather than encrypting the whole message (which is costly), a digest of the message can be encrypted — the result is a **digital signature** of the message. The message is sent in the clear along with this digital signature. The result is only the person with the private key who could have "signed" the digest (the digest is authentic; non-repudiation). Given a digest, it is computationally difficult for someone to forge a message (irreversible hash function for the digest). However, this remains open to a birthday attack – when signing a document, always add something to affect the digest thus not allowing someone else to have a pre-pared other message with matching digest.

NOTE The **Digital Signature Algorithm** (DSA) is a way of developing a digital signature; based on El-Gamal where a number system is chosen, a representation of it is chosen – a generator, and then a power value is selected. The public key and private keys are determined from these choices. Digital signatures could be developed using an El-Gamal approach such as here, or even with the RSA method.

Securing E-mail

If Alice wants to send a confidential e-mail m to Bob, Alice would generate a random *symmetric* private key K_s; the message is encrypted with this key (for efficiency) and also encrypts it with Bob's public key. Both $K_s(m)$ and $K_B(K_s)$ are sent to Bob. Bob then uses his private key to decrypt and recover K_s which he uses to recover m by decrypting $K_s(m)$. This only provides **confidentiality**.

If Alice wants to provide sender **authentication** message integrity, Alice would digitally sign the message and send both the message (in the clear) and digital signature. If she wants to further provide secrecy, sender authentication, *and* message integrity, she would use three keys: her private key, Bob's public key, and a newly-created symmetric key.

Secure Socket Layer (SSL)

Transport Layer Security (TLS), through **Secure Socket Layer** (SSL), was privately developed by Netscape in 1993. It was implemented in web browsers (with libraries available) and designated URLs for HTTP over SSL as `https`. It offers: **confidentiality**, **data integrity**, and **authentication** (server mandatory, client optional — e.g., for corporate companies). SSL operates conceptually like a (pseudo-session layer) between the application and transport layers.

NOTE The TCP connection/handshake must be cemented firmly before SSL begins its handshake process.

By design, SSL possesses the following properties (see slide on SSL Handshake):

- automatic **authentication** of the server (exchange certificates in initial handshake),
- **efficient encryption of data** (negotiate a symmetric key using asymmetric encryption in the handshake — actually want multiple keys [x4; messages and signatures on both sides]; use the best encryption algorithm available to client and server — negotiate in handshake),
- **verification of data integrity and order** (include a MAC, encrypted with the data and include sequence numbers — TCP sequence number still guarantees the delivery order and encrypted SSL sequence number ensures nobody is tampering with the sequence).
- **resistant to replay attacks** (using encrypted sequence numbers and using server- and client-side nonces in set-up),

Secure Socket Layer (SSL) continued on next page...

- **resistant to connection stealing** (explicitly closing connections, including nonces in set-up until symmetric key is established), and
- **resistant to changes in unencrypted handshake information** (confirming the set-up conversation, the set of all unencrypted messages, with a digest).

Best practice dictates that you use different keys for encryption and message authentication codes. Using the same key in different algorithms or in both encryption/decryption modes opens another set of attacks to discover the key. SSL uses a **key derivation function** (KDF) to generate 4 keys from a master key: K_c – encrypt data from client to server, M_c – generate MAC for data from client to server, K_s – encrypt data from server to client, and M_s – generate MAC for data from server to client.

NOTE Data exchange in TCP is analogous to MPEG4 compression: the splitting of blocks into data where MACs are chosen at key points (have not sent in a while or fixed-sized data). Consider all of this security a problem of risk management: how sensitive is the data and how much effort is willing to be done?

DATA EXCHANGE. The problem here is that TCP treats all data as a **stream**. We want to keep this conceptually for the application layer. As a stream of data, we do not know when it ends — when can we compute a MAC for data integrity? The solution here is to split the stream into variable-sized blocks of data which a MAC can be applied to: a **length** portion, the **data** portion, and then the **MAC**.

SSL sequence numbers count the blocks of data covered by a MAC. TCP already guarantees that the data *should* arrive in correct order, so the sequence number should be predictable. These sequence numbers are for **data integrity** only — the sequence numbers are included in the computation of the MAC and do not have to be explicitly transmit the number thanks to TCP's reliability.

Therefore, the MAC is a hash of M_x secret key ($x = c$ or s), a sequence number, and the data.

Note that the client and server *must* agree on the encryption algorithms. The client lists all of the algorithms that it can accept as part of the handshake. The server selects one and informs the client of the selection. Common algorithms include: DES, 3DES, RC2, RC4, AES, RSA, Diffie-Hellman, and Fortezza. *What prevents an attacker from just changing the list of algorithms from the client or selection of the server*? Before finishing the handshake, client and server exchange a MAC each of the set of all handshake messages.

The client and server exchange data (a pre-master key) from which all other encryption keys are derived; the **pre-master key** is combined with a client-side and a server-side nonce with a pseudo-random number generator to get a master key; two nonces ensure that a replay attack will fail because either the client or server will use a newly-generated nonce for a new connection. The **master key** is combined with new nonces to generate a "key block". Elements of this key block are used to define all keys and client and server initialization.

Finally, as messages are traded off, a message type is included in the SSL message (e.g., regular data, connection close, change cipher algorithm request, handshake messages) and a version number. See slide for **SSL handshake**.