Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science

6.829 Fall 2002 **Lecture 2: The Internetworking Problem** September 10, 2002

Overview. This lecture looks at the issues that arise in internetworking different networks together. We will study the Internet design principles, IP, the Internet's best-effort service model, its design and protocol standardization philosophy, and IP.

We then discuss the split between IP and TCP, and study how TCP achieves reliable data transport. We will look at TCP's cumulative acknowledgment feedback, and two forms of retransmission upon encountering a loss: *timer-driven* and *data-driven* retransmissions. The former relies on estimating the connection round-trip time (RTT) to set a timeout, whereas the latter relies on the successful receipt of later packets to initiate a packet recovery without waiting for a timeout. Examples of data-driven retransmissions include the *fast retransmission* mechanism and *selective acknowledgment (SACK)* mechanism.

Readings: CK74, Cla88.

1 The Internetworking Problem: Many Different Networks

In the previous lecture, we discussed the principles underlying packet switching, and developed a scheme for connecting local-area networks (LANs) together. The key idea was a device called a switch, which forwarded packets between different LANs.

While such networks work and are common, they are not enough to build a global data networking infrastructure. They have two main failings:

1. *They don't accommodate heterogeneity.*
2. *They don't scale well to large networks.*

Each point bears some discussion. Although commonplace, Ethernets are by no means the only network link technology in the world. Indeed, even before Ethernets, packets were being sent over other kinds of links including wireless packet radio, satellites, and telephone wires. Ethernets are also usually ill-suited for long-haul links that have to traverse hundreds of miles. Advances in link technologies of various kinds has always been occurring, and will continue to occur, and our goal is to ensure that *any* new network link technology can readily be accommodated into the global data networking infrastructure. This is the goal of link heterogeneity.

The LAN switching solution we studied in Lecture 1 requires switches to store per-host (actually, per-interface) state in the network; in the absence of such state for a destination, switches revert to broadcasting the packet to all the links in the network. This does not scale to more than a moderate number of hosts and links.

The internetworking problem may be stated as follows: Design a scalable network infrastructure that interconnects different smaller networks, to enable packets to be sent between hosts across networks of very different performance characteristics.

Some examples of the differing characteristics include:

- **Addressing.** Each network might have its own way of addressing nodes; for example, Ethernet's use a 6-byte identifier while telephones use a 10-digit number.
- **Bandwidth & latency.** Heterogeneity in bandwidths range from a small number of bits/s (e.g., power lines) to many Gigabits/s, spanning many orders of magnitudes. Similarly, latencies can range from microseconds to several seconds.
- **Packet size.** The maximum packet sizes in general will vary between networks.
- **Loss rates.** Networks differ widely in the loss rates and loss patterns of their links.
- **Packet routing.** Packet routing could be handled differently by each constituent network, e.g., packet radio networks implement different routing protocols from wireline networks.

Our scalability goal is to reduce the state maintained by the switches and the bandwidth consumed by control traffic (e.g., routing messages, periodic broadcasts of packets/messages, etc.) as much as we can.

1.1 Gateways

Communication between different networks in an internetwork is facilitated by entities called *gateways.* Gateways interface between two networks with potentially differing characteristics and move packets between them. There are at least two ways of interfacing between different networks: *translation* and a *unified network layer.*

Translation-based gateways work by translating packet headers and protocols over one network to the other, trying to be as seamless as possible. Examples of this include the OSI X.25/X.75 protocol family, where X.75 is a translation-based "exterior" protocol between gateways.

There are some major problems with translation.

- **Feature deficiency.** As network technologies change and mature and new technologies arise, there are features (or bugs!) in them that aren't preserved when translated to a different network. As a result, some assumption made by a user or an application breaks, because a certain expected feature is rendered unusable since it wasn't supported by the translation.
- **Poor scalability.** Translation often works in relatively small networks and in internetworks where the number of different network types is small. However, it scales poorly because the amount of information needed by gateways to perform correct translations scales proportional to the square of the number of networks being inter-connected. This approach is believed to be untenable for large internetworks. (As an aside, it's worth noting that the widespread deployment of NATs is making portions of the Internet look a lot like translation gateways!)

1.2 Design Principles

The designers of the Internet (the "big I" Internet!) protocols recognized the drawbacks of translation early on and decided that the right approach was to standardize some key properties across all networks, and define a small number of features that all hosts and networks connected to the Internet must implement.

Over time, we have been able to identify and articulate several important design principles that underlie the design of Internet protocols. We divide these into *universality* principles and *robustness* principles, and focus on only the most important ones (this is a subjective list).

Universality principles

U1. IP-over-everything. All networks and hosts must implement a standard network protocol called IP, the Internet Protocol. In particular, all hosts and networks, in order to be reachable from anywhere, must implement a standard well-defined addressing convention. This removes a scaling barrier faced by translation-based approaches and makes the gateways simple to implement.

U2. Best-effort service model. A consequence of the simple internal architecture of the gateways is that the service model is best-effort. All packets are treated (roughly) alike and no special effort is made inside the network to recover from lost data in most cases. This well-defined and simple service model is a major reason for the impressive scalability of the architecture.

U3. The end-to-end arguments. End-to-end arguments, articulated long after the original design of TCP/IP, permeates many aspects of the Internet design philosophy. Its layered architecture keeps the interior of the network simple and moves all the complicated machinery for reliability, congestion control, etc. to the end-hosts where they can be implemented more completely. Observe that the original design (CK74) had TCP and IP combined both at the end-points and in the gateways. The separation between the two, with TCP functionality being removed from the gateways, was an important step in the evolution of the Internet.

Robustness principles

R1. Soft-state inside the network. Most of the state maintained in the gateways is *soft-state*, which is easily refreshed when it is lots or corrupted. Routing table state is a great example of this; gateways use these tables to decide how to route each packet, but the periodic refreshes in the form of routing updates obviate the need for complex error handling software in the routers. In this way, gateways and routing protocols are designed to overcome failures as part of their *normal* operation, rather than requiring special machinery when failures occur.

R2. Fate sharing. When a host communicates with another, state corresponding to that communication is maintained in the system. In the Internet, the critical state for this is shared between the two (or more) communicating entities and not by any other entities in the network. If one of the end hosts fails, so do the others; in this way they share fate. If a gateway fails en route, the end-to-end communication doesn't—the state in the gateway is soft and the gateway doesn't share fate with the end hosts. This is in stark contrast with the OSI X.25 approach where gateways maintained hard connection state and end hosts shared fate with them.

R3. Conservative-transmission/liberal reception. "Be conservative in what you send; be liberal in what you accept." This guideline for network protocol design significantly increases the robustness of the system. This principle is especially useful when different people, who may have never spoken to each other, code different parts of a large system. This is important even when writing code from a specification, since languages like English can be ambiguous, and specifications can change with time. A simple example of this principle is illustrated in what a TCP sender would do if it saw an acknowledgment that acknowledged a packet it had never sent. The worst behavior is to crash because it isn't prepared to receive something it didn't expect. This is bad—what's done is instead is to silently drop this ACK and see what else comes from the peer. Likewise, if a sender transmits a packet with a non-standard option, it should do so only if it has explicitly been negotiated.

1.3 Weaknesses

The Internet architecture is not without its weaknesses, many of which are a consequence of its original design goals (see, e.g., [Cla88], Clark's 1988 retrospective paper).

- The architecture fundamentally relies on the trustworthiness of end-systems. It does not consider the possibility of malicious end systems and how to build a trusted network despite this.
- Greedy sources aren't handled well (non-cooperating users, buggy or malicious implementations). TCP does a good job of sharing bandwidth, but more and more traffic isn't TCP.
- Security issues, for a long time, weren't considered as paramount as they are these days.
- Weak accounting and pricing tools. One of the original goals, but hasn't developed much.
- Administration and management tools are not particularly mature.
- Incremental deployment. Not really a fundamental "weakness," but has to be recognized by every protocol designer.

1.4 Standards process

An interesting aspect of the continuing development of the Internet infrastructure is the process by which standards are set and developments are made. By and large, this has been a very successful exercise in social engineering, in which people from many different organizations make progress toward consensus. The Internet Engineering Task Force (IETF), a voluntary organization, is the standards-setting body. It meets every 4 months, structured around short-lived working groups (usually less than 2 years from inception to termination). Internet standards are documented in RFCs ("Request for Comments" documents) published by the IETF (several different kinds of RFCs exist), and documents in draft form before they appear as RFCs are called "Internet Drafts." Check out `http://www.ietf.org/` for more information.

Working groups conduct a large amount of work by email, and are open to all interested people. In general, voting on proposals are shunned, and the process favors "rough consensus." Running, working code is usually a plus, and can help decide direction. It used to be the case that at least two independent, interoperating implementations of a protocol were required for a standard to be in place, but this seems to be generally ignored these days.

2 Digression: The original TCP/IP

Cerf & Kahn describe their solutions to the internetworking problems in their seminal 1974 paper. This early work has clearly had enormous impact, and this is one of the main reasons we study it despite the actual details of today's IP being very different. Some of the design principles have been preserved to date. Furthermore, this is a nice paper in that it presents a design with enough detail that it feels like one can sit in front of a computer and code up an implementation from the specifications described!

The following are the key highlights of their solution:

- Key idea: **gateways.** Reject translation in favor of gateways.
- IP over everything—beginnings of protocol hourglass model evident in first section of paper.
- As a corollary, addressing is common to entire internetwork. Internal characteristics of networks are of course different.
- Standard packet header format. But notice mixing of transport information in routing header. (And some fields like ttl are missing.)
- Gateways perform *fragmentation* if needed, *reassembly* is done at the end hosts.
- **TCP:** Process-level communication via the TCP, a reliable, in-order, byte-stream protocol. And in fact, they had the TCP do reassembly of segments (rather than IP that does it today at the receiver). TCP and IP were tightly inter-twined together; it took several years for the TCP/IP split to become visible.
- The spent a lot of time on figuring out how multiple processes between the same end hosts communicate. They considered two options:
  1. Use same IP packet for multiple TCP segments.
  2. Use different packets for different segments.

  They didn't consider the option of many TCP connections between the same hosts!
- 2 is simpler than 1 because it's easier to demultiplex and out-of-order packets are harder to handle in 1. [Under what conditions might 1 be better?]
- Demux key is port number (typically one process per port).
- Byte-stream TCP, so use ES and EM flags. EM, ES, SEQ and CT adjusted by gateways. I.e., gateways look through and modify transport information. End-to-end checksums for error detection.
- Sliding window, cumulative ACK-based ARQ protocol (with timeouts) for reliability and flow control. Similar to CYCLADES and ARPANET protocols.
- Window-based flow control. Also achieve process-level flow control.
- Implementation details/recommendations for I/O handling, TCBs.
- First-cut connection setup and release scheme. (Heavily modified subsequently to what it is today.)

- Way off the mark on accounting issues! (The problem is a lot harder than they anticipated.)
- One-line summary: A great paper that details an internetworking protocol of great significance. Enough details to actually implement it, but not a dry document to read!

3 Today's TCP/IP: IPv4

The most important thing to note about IP and its addressing strategy is that it is designed for scalability. This is achieved using a simple idea: IP addresses aren't arbitrary; rather, they signify *location* in the network topology. This allows addresses to be aggregated and maintained in routing tables, essentially achieving a hierarchical structure. Aggressive aggregation allows the amount of routing table state to scale well as the size of the network increases.

3.1 Addressing

When version 4 of IP was standardized in RFC 791 in 1981, addresses were standardized to be 32-bits long. At this time, addresses were divided into classes, and organizations could obtain a set of addresses belonging to a class. Depending on the class, the first several bits correspond to the "network" identifier and the others to the "host" identifier. *Class A* addresses start with a "0", use the next 7 bits of network id, and the last 24 bits of host id (e.g., MIT has a Class A net, 18). *Class B* addresses start with "10" and use the next 14 bits for network id and the last 16 bits for host id. *Class C* addresses start with "110" and use the next 21 bits for network id, and the last 8 bits for host id. Class D addresses are used for IP multicast; they start with "1110" and use the next 28 bits for the group address. Addresses that start with "1111" are reserved for experiments.

This two-level network-host hierarchy quickly proved insufficient and in 1984 a third hierarchical level corresponding to "subnets" was added. Subnets can have any length and are specified by a 32-bit network "mask"; to check if an address belongs to a subnet, take the address and zero out all the bits corresponding to zeroes in the mask; the result should be equal to the subnet id for a valid address in that subnet. The only constraint on a valid mask is that the 1s must be contiguous from most significant bit, and all the 0s must be in the least significant bits.

In the early 1990s, IPv4 addresses started running out. In a great piece of "just-in-time" engineering, the IETF deployed CIDR (pronounced "Cider" with a short "e"), or Classless Inter-Domain Routing recognizing that the division of addresses into classes was inefficient and that routing tables were exploding in size because more and more organizations were using non-contiguous Class C addresses. The main problem was that Class B addresses were getting exhausted; these were the most sought after because Class A addresses required great explanation to the IANA (Internet Assigned Numbers Authority) because of the large (2^{24}) host addresses, which Class C addresses with just 256 hosts per network were grossly inadequate. Because only $2^{14} = 16384$ Class B networks are possible, they were running out quickly.

CIDR optimizes the common case. The common case is that while 256 addresses are insufficient, most organizations require at most a few thousand. Instead of an entire Class B, a few Class C's will suffice. Furthermore, making these *contiguous* will reduce routing table sizes because routers aggregate routes based on IP prefixes in a classless manner. We will discuss routing table aggregation in more detail when we discuss routing protocols in a few weeks.

3.2 Fragmentation and Reassembly

Different networks do not always have the same MTU, or Maximum Transmission Unit. When an IP gateway receives a packet[1] that needs to be forwarded to a network with a smaller MTU than the size of the packet, it can take one of several actions.

1. Discard the packet. In fact, IP does this if the sender set a flag on the header telling gateways not to fragment the packet. After discarding it, the gateway sends an error message using the ICMP protocol to the sender. [What are some reasons for providing this feature in the Internet?]
2. Fragment the packet. The default behavior in IPv4 is to fragment the packet into MTU-sized fragments and forward each fragment to the destination. There's information in the IP header about the packet ID and offset that allows the receiver to reassemble fragments.

3.3 Time-to-live (TTL)

To prevent packets from looping forever, the IP header has a TTL field. It's usually decremented by 1 at each router and the packet is discarded when the TTL is zero.

3.4 Type-of-service (TOS)

The IP header includes an 8-bit type-of-service field that allows routers to treat packets differently. It's largely unused today, although we'll later see how differentiated services intends to use this field.

3.5 Protocol field

To demultiplex an incoming packet to the appropriate higher (usually transport) layer, the IP header contains an 8-bit protocol field.

3.6 Header checksum

To detect header corruption, IP uses a (weak) 16-bit header checksum.

3.7 IP options

IP allows nodes to introduce options in its header. A variety of options are defined but are hardly used, even when they're useful. This is because it's expensive for high-speed routers (based on the way they're designed today) to process options on the fast path. Furthermore, most IP options are intended for end-points to process, but IPv4 mandates that all routers process all options, even when all they have to do is ignore them. This makes things slow—and is a severe disincentive against options. Engineering practice today is to avoid options in favor of IP-in-IP encapsulation or, in many cases, having the routers peek into transport headers (e.g., the TCP or UDP port numbers) where the fields are in well-known locations and require little parsing.

[1] Or *datagram.*

4 Today's TCP/IP: TCP

4.1 The Problem: A Best-Effort Network

A best-effort network greatly simplifies the internal design of the network, but implies that there may be times when a packet sent from a sender does not reach the receiver in a timely manner. TCP deals with three specific problems:

1. *Losses* occur, typically because of congestion or packet corruption.
2. Packet delays are *variable*.
3. Packet *reordering* occurs, typically because of multiple routes between end-points, or pathologies in router implementations.

Many applications, including bulk file transfers, the Web, etc. require reliable data delivery, with packets arriving *in-order*, in the same order in which the sender sent them. These applications will benefit from TCP.

4.2 The TCP Service Model

The TCP service model is that of an in-order, reliable, duplex, byte-stream abstraction. It doesn't treat datagrams as atomic units, but instead treats bytes as the fundamental unit of reliability. The TCP abstraction is for *unicast* transport, between two hosts (actually between two *interfaces*). Duplex refers to the fact that the same connection handles reliable data flow in both directions.

In general, reliable transmission protocols can use one or both of two kinds of techniques to achieve reliability in the face of packet loss. The first, called *forward error correction* (FEC), is to use redundancy in the packet stream to overcome the effects of some losses. The second is called *automatic repeat request* (ARQ), and uses packet retransmissions. The idea is for the sender to infer the receiver's state using acknowledgments (ACKs) it gets, and determine that packets are lost if an ACK hasn't arrived for a while, and retransmit if necessary.

In TCP, the receiver periodically[2] informs the sender about what data it has received via ACKs. TCP ACKs are *cumulative.* For example, the sequence of bytes:

```
1:1000 1001:1700 2501:3000 3001:4000 4001:4576
```

received at the receiver will cause the acknowledgments

```
1001 1701 1701 1701 1701
```

to be sent by the receiver after each datagram arrival. Each ACK acknowledges all bytes received in-sequence thus far and tells the sender what the next expected in-sequence byte is.

Each TCP ACK includes in it a receiver-advertised window that tells the sender how much space is available in its socket buffer at any point in time. This is used for end-to-end *flow control.* This is not

[2]Every time it receives a datagram; modern TCP receivers implement a *delayed ACK* policy where they should ACK only on every other received datagram as long as data arrives in sequence, and must ack at least once every 500ms. BSD-derived implementations use a 200ms "heartbeat" timer for this.

to be confused with *congestion control,* which is how resource contention for "inside-the-network" router resources (bandwidth, buffer space) is dealt with. Flow control only deals with making sure that the sender doesn't overrun the receiver at any stage (it's a lot easier than congestion control, as you can see). We will study this in more detail in later lectures.

TCP has two forms of retransmission upon encountering a loss: *timer-driven* retransmissions and *data-driven* retransmissions. The former relies on estimating the connection round-trip time (RTT) to set a timeout value; if an ACK isn't received within the timeout duration of sending a segment, that segment is retransmitted. On the other hand, the latter relies on the successful receipt of later packets to initiate a packet recovery without waiting for a timeout.

4.3 TCP timers

To perform retransmissions, the sender needs to know when packets are lost. If it doesn't receive an ACK within a certain amount of time, it assumes that the packet was lost and retransmits it. This is called a *timeout* or a *timeout-triggered* retransmission. The problem is to determine how long the timeout period should be.

What factors should the timeout depend on? Clearly, it should depend on the connection's round-trip time (RTT). To do this, the sender needs to estimate the RTT. It obtains samples by monitoring the time difference between sending a segment and receiving a positive ACK for it. It needs to do some averaging across all these samples to maintain a (running) average. There are many ways of doing this; TCP picks a simple approach called the Exponential Weighted Moving Average (EWMA), where $srtt = \alpha \times r + (1 - \alpha) srtt$. Here, r is the current sample and $srtt$ the running estimate of the *smoothed* RTT. $\alpha = 1/8$ in TCP for efficient computability (can be implemented in fixed point using bit-shifts).

We now know how to get a running average of the RTT. How do we use this to set the retransmission timeout, or RTO? One option, an old one, is to pick a multiple of the smoothed RTT and use it. For example, we might pick $RTO = \beta * srtt$, with β set to a constant like 2. In fact, the original TCP specification (RFC 793) used precisely this method. Unfortunately, this simple choice doesn't work too well in preventing *spurious* retransmissions. Spurious retransmissions are those done for segments still in transit but presumed lost by the sender. This could lead to bad congestion effects, because the principle of "conservation of packets" will no longer hold true.

A nice and simple fix for this is to make the RTO a function of both the average and the standard deviation. Under nice Gaussian assumptions of delay samples (not true in practice, but it still works), the tail probability of a spurious retransmission when the RTO is a few standard deviations away from the mean is rather small. So, TCP uses an RTO set according to: $RTO = srtt + 4 \times rttvar$, where $rttvar$ is the (inappropriately named) mean linear deviation of the RTT from its mean. I.e., $rttvar$ is calculated as $rttvar = \gamma \times dev + (1 - \gamma) \times rttvar$, where $dev = |r - srtt|$ and $\gamma = 1/4$.

This isn't the end of the story. TCP also suffers from a significant *retransmission ambiguity* problem. When an ACK arrives for a segment the sender has retransmitted, how does the sender know whether the RTT to use is for the original transmission or for the retransmission? This might seem like a trivial detail, but in fact is rather vexing because the RTT estimate can easily become meaningless and throughput can suffer. The solution to this problem is surprisingly simple—ignore samples that arrive when a retransmission is pending or in progress. I.e., don't consider samples for any segments that have been retransmitted.

The modern way of avoiding the ambiguity problem is to use the TCP *timestamp option.* Most good TCP implementations follow RFC 1323's recommendation of using the timestamp option. Here, the sender uses 8 bytes (4 for seconds, 4 for microseconds) and stamps its current time on the segment. The receiver, in the cumulative ACK acknowledging the receipt of a segment simply *echoes* the sender's stamped value, and now the sender can do the calculation trivially by subtracting the echoed time in the ACK from the current time. Note that it now doesn't matter if this was a retransmission or an original transmission; the timestamp effectively serves as a "nonce" for the segment.

The other important issue is deciding what happens when a retransmission times out. Obviously, because TCP is a "fully reliable" end-host protocol, the sender must try again. But rather than try at the same frequency, it takes a leaf out of the contention resolution protocol literature (e.g., Ethernet CSMA) and performs *exponential backoffs* of the retransmission timer.

A final point to note about timeouts is that they are extremely *conservative* in practice. TCP retransmission timers are usually (but not always) coarse, with a granularity of 500 or 200 ms. This is a big reason why spurious retransmissions are rare in modern TCPs, but also why timeouts during downloads are highly perceptible by human users.

4.4 Fast retransmissions

Because timeouts are expensive (in terms of killing throughput for a connection, although they are a necessary evil from the standpoint of ensuring that senders back-off under extreme congestion), it makes sense to explore other retransmission strategies that are more responsive. Such strategies are also called *data-driven* (as opposed to *timer-driven*) retransmissions. Going back to the earlier example:

```
1:1000 1001:1700 2501:3000 3001:4000 4001:4576
```

with ACKs

```
      1001      1701      1701      1701      1701
```

It's clear that a sequence of repeated ACKs are a sign that something strange is happening, because in normal operation cumulative ACKs should monotonically increase. Repeated ACKs in the middle of a TCP transfer can occur for three reasons:

1. Window updates. When the receiver finds more space in its socket buffer, because the application has read some more bytes, it can send a window update even if no new data has arrived.

2. Segment loss.

3. Segment reordering. This could have happened, for example, if datagram 1701-2500 had been reordered because of different routes for different segments.

Repeated ACKs that aren't window updates are called *duplicate* ACKs or *dupacks.* TCP uses a simple heuristic to distinguish losses from reordering: if the sender sees an ACK for a segment more than three segments larger than a missing one, it assumes that the earlier (unacknowledged) segment has been lost. Unfortunately, cumulative ACKs don't tell the sender which segments have reached; they only tell the sender what the last in-sequence byte was. So, the sender simply *counts* the number of dupacks and infers that if it see three or more dupacks, that the corresponding segment was lost. Various empirical studies have shown that this heuristic works pretty well, at least on today's Internet, where reordering TCP segments is strongly discouraged and there isn't much "parallel" ("dispersity") routing.

4.5 Selective acknowledgment (SACK)

When the bandwidth-delay product of a connection is large, e.g., on a high-bandwidth, high-delay link like a satellite link, windows can get pretty large. When multiple segments are lost in a single window, TCP usually times out. This causes abysmal performance for such connections, which are colloquially called "LFNs" (for "Long Fat Networks" and pronounced "elephants," of course). Motivated in part by LFNs, selective ACKs (SACKs) were proposed as an embellishment to standard cumulative ACKs. SACKs were standardized a few years ago by RFC 2018, after years of debate.

Using the SACK option, a receiver can inform the sender of up to three maximally contiguous blocks of data it has received. For example, for the data sequence:

```
1:1000 1001:1700  2501:3000  3001:4000 4577:5062 6000:7019
```

received, a receiver would send the following ACKs and SACKs (in brackets):

```
      1001      1701         1701         1701       1701         1701
                         [2501-3000]  [2501-4000] [4577-5062; [6000-7019;
                                                   2501-4000]   4577-5062;
                                                                2501-4000]
```

SACKs allow LFN connections to recover many losses in the same window with a much smaller number of timeouts. While SACKs are in general a Good Thing, they don't always prevent timeouts, including some situations that are common in the Internet today. One reason for this is that on many paths, the TCP window is rather small, and multiple losses in these situations don't give an opportunity for a TCP sender to use data-driven retransmissions.

4.6 Some other issues

There are a few other issues that are important for TCP reliabilty.

1. **Connection setup/teardown.** At the beginning of a connection, a 3-way handshake synchronizes the two sides. At the end, a teardown occurs.

2. **Segment size.** How should TCP pick its segment size for datagrams? Each side picks a "default" MSS (maximum segment size) and exchanges them in the SYN exchange at connection startup; the smaller of the two is picked.

   The recommended way of doing this, to avoid potential network fragmentation, is via *path MTU discovery.* Here, the sender sends a segment of some (large) size that's smaller than or equal to its interface MTU (when the IP and link-layer headers are added) with the “DON’T FRAGMENT (DF)” flag in the IP header. If the receiver gets this segment, then clearly every link en route could support this datagram size because there was no fragmentation. If not, and an ICMP error message was received by the sender from a router saying that the packet was too big and would have been fragmented, the sender tries a smaller segment size until one works.

3. **Low-bandwidth links.** Several TCP segments are small in size and are mostly comprised of headers; examples include telnet packets and TCP ACKs[3]. To work well over low-bandwidth links, TCP header compression can be done (RFC 1144). This takes advantage of the fact that most of the fields in a TCP header are either the same or are predictably different from the previous one. This allows a 40-byte TCP+IP header to be reduced to as little as 3-6 bytes.

5 Summary

The Internet architecture is based on design principles that provide universality and improve robustness to failures. It achieves scaling by using an addressing structure that reflects network topology, enabling topological address information to be aggregated in routers.

TCP provides an in-order, reliable, duplex, byte-stream abstraction. It uses timer-driven and data-driven retransmissions; the latter provides better performance under many conditions, while the former ensures correctness.

[3]TCP does allow data to be piggybacked with ACKs and most data segments have valid ACK fields acknowledging data in the opposite direction of the duplex connection.